for the B.B.C. Micro

Welcome FORTH_

By J.W. BROWN

for HCCS

CONTENTS PAGE

Introduction

This manual has been produced as an introductory course to FORTH on the BBC Microcomputer. Most of the contents however, will be equally applicable to other implementations on other machines. For BBC users, the manual should be used in conjunction with the documentation supplied with the program ROM, as this contains the '79 FORTH glossary and explanations/examples of the extra definitions for the BBC machine. The manual is not intended to be definitive, but merely an introduction, to this cryptic but very powerful language, and it is the authors' intention to point the user in the correct direction, rather than lead him. Included are many practical examples, both graphic and otherwise, the final example being that of a Mnemonic Assembler, together with an experimental "Active" label.

FORTH FOR THE BBC MICROCOMPUTER

APPLICATION FOR THE IMPLEMENTATION CAN BE MADE TO THE AUTHORS SOLE AGENTS

HCCS ASSOCIATES
533 DURHAM RD.,
LOW FELL
GATESHEAD
TYNE & WEAR.
TEL. 0632-821924

Acknowledgements

The author would like to thank the following without whom the production of this document would not have been possible.

Jim Golightly and Alan Heslop of Mrrs. HCCS Associates, whose enthusiasm prompted this implementation of FORTH for the BBC Micro. Diane and Babs for doing all the typing from my original scribble. Jim Moir-Howes for checking that my examples actually worked. Various members of the Forth Interest Group for several of the application examples.

WELCOME FORTH

FORTH was created in 1969 by Charles H. Moore as a convenient method of controlling machinery by computer. Existing high-level languages such as BASIC being totally unsuited because of their slow speed, the other alternative - machine-code is both difficult to write and instal quickly.

FORTH has the benefit of the relative ease of writing a high-level language and a very fast execution speed. This is achieved because FORTH is a compiled language - its final form in memory being a "thread" of pointers to machine-code routines.

FORTH is both extensible and interactive, new words for applications can be created and tested immediately and their actions changed, if required, until the word does exactly what you require of it. Once a word has been created, it becomes part of FORTH itself, i.e. you may use that word in the creation of others.

Writing a FORTH program begins with specifying first of all the overall action you wish the program to perform. This can be given a name, then this action is broken down into sub-actions or tasks, each performing a logical part of the whole program. These to, in large applications, may be further sub-divided into well defined procedures until the tasks themselves are supported and already defined in the FORTH nucleus dictionary. Because of this structured approach to programming, generally speaking it is difficult to write an unstructured program in FORTH, ardent fans of GOTO's in BASIC will no doubt be disappointed, there are none in FORTH.

This introduction to FORTH is based on the public domain Publications provided through the courtesy of the

Forth Interest Group
P.O. Box 1105

San Carlos U.S.A.

In this manual, all FORTH words are displayed in UPPER CASE CHARACTERS, as this, generally speaking, is how they are typed at the keyboard. As there may be occasional confusion between some FORTH words and punctuation in this text in the explanation of the words, such FORTH words are enclosed in square brackets (these of course should not be typed at the keyboard!).

All keyboard input whether text or numeric must be terminated by a carriage return, and in the examples that follow, the output from the machine is underlined for clarity.

The numeric output/input, unless stated is in DECIMAL base.

In this implementation a FORTH word can be any length to a maximum of 31 characters, it should be noted at this stage that a space is used in FORTH as an unconditional delimeter, and that FORTH words therefore cannot contain a space. Examples of valid words are:-

VOCABULARY

FORTH

<TABLE!

EMPTY-BUFFERS

The following examples are not valid and will result in an error flag as the two words will be sought for separately.

TO TABLE (includes space - FORTH will treat this a two separate words)

FILL-TABLE-WITH-STACK-CONTENTS-STARTING-FROM-TOP (This word is over 31 characters in length)

Any characters available at the keyboard can be used in a FORTH word, including numbers, e.g.

TASK1 2+

A FORTH "screen" in this implementation is composed of 1024 bytes. This gives 16 lines of 64 characters, hence the maximum characters per line (before a carriage return) is 64 characters.

After a COLD start you will be rewarded with:-

JWB-FORTH V2

Now type the following at the keyboard followed by a CR, LEAVING A SPACE BETWEEN EACH WORD

: PLUSES BEGIN 43 EMIT SPACE AGAIN ;

OK

OK is the standard response to all correct entries in FORTH. What has happened is that a new word has been entered into the dictionary. Now type

PLUSES (CR)

The screen will instantly fill up with pluses - forever if you wish! As this definition does not contain a conditional to leave it, the endless loop BEGIN.....AGAIN will do just that - the words between BEGIN and AGAIN will execute forever, so press the BREAK key and re-enter FORTH with:

WARM - again you should have the following on the screen

JWB-FORTH V2

Now type VLIST (CR)

The FORTH dictionary will be displayed on the screen, the display can be discouraged from scrolling by typing ESC, when the word PLUSES will be seen at the very top of the dictionary.

Try typing FORGET PLUSES OK

Then type PLUSES

FORTH will respond with:- PLUSES ?MSG # 0

Message 0 is the error code for entry not found, in other words FORTH has forgotten your word PLUSES. Typing VLIST again will indeed show that the word PLUSES no longer exists. (It wasn't much use anyway!)

The original start is a COLD start to FORTH and after you typed BRK, WARM gave a WARM start - remembering any applications you had entered after the COLD start.

Wherever either a COLD or WARM start is entered, FORTH is initialised as follows

The NUMERIC BASE IS DECIMAL

The CURRENT VOCABULARY IS FORTH

The CONTEXT VOCABULARY IS FORTH

The Return stack is initialised and the computation stack is cleared.

In most high-level languages the stacks (areas of memory used to store addresses, data etc.) are usually well out of sight, but in FORTH these are totally accessible to the user.

The RETURN stack is the normal 6502 stack, on which return addresses, of routines, data put there with instructions such as the machine-code PHA etc. are stored.

The COMPUTATION stack is the FORTH work area, on which most calculations are performed, it is a LAST IN FIRST OUT STACK (LIFO), so that data put on it last is the most accessible, almost all FORTH words are stack operators i.e. have some direct or indirect action on data stored on the stack.

[.] is used to print out the contents of the TOP of the stack in single-precision form - typing it when the stack is empty will give the response

0 . ?MSG # 1

Where error message 1 is the code for STACK EMPTY

Now try the following first making sure the stack is empty by typing [.] until the above message is obtained (Don't forget the spaces!)

1 2 3 4 5 (CR) OK

. . . . . . (CR) 5 4 3 2 1 OK

Immediately you will notice the last number typed is the first one printed. This adequately demonstrates the action of the LIFO stack. Now try this:-

24 12 + . 36 OK

Note that in this addition, the plus operator has been inserted LAST, this is post-fix notation, used in many modern calculators and may take some getting used to. However, having mastered it, the user will find it far easier to use, especially in complex expressions which would normally contain many brackets.

24 12 - . 12 OK
24 12 / . 2 OK
24 12 * . 288 OK
-15 12 + . -3 OK
-56 23 - . -79 OK

The above are examples of SINGLE PRECISION OPERATIONS. In FORTH unsigned numbers are stored in twos - complement form in the range -32768 to +32767 inclusive. All FORTH numbers are in integer form rather than floating-point form. This is really no problem, most calculations can be solved with FORTH's integer operations. The above demonstration shows that ALL arithmetic operators expect to find the necessary values ALREADY on the stack - hence the use of the post-fix notation.

F.P. to be included in extension ROM.

In the following and subsequent table of stack action the format (n1/n2/........

shows the values on the stack BEFORE operations and
.... (exp))
after an operation has been carried out.

SINGLE PRECISION OPERATORS

WORD	DESCRIPTION	
+	ADD	(n1/n2 ..... sum nl+n2)
-	SUBTRACT	(n1/n2 ..... diff.n1-n2)

WORD	DESCRIPTION
*	MULTIPLY (n1/n2 ..... Product n1*n2)
/	DIVIDE (n1/n2 ..... quotient of n1/n2)
MOD	REMAINDER (n1/n2 ..... remainder of n1/n2)
/MOD	LEAVE QUOTIENT WITH REMAINDER BENEATH (n1/n2 ..... rem/quotient)
*/	MULTIPLY DIVIDE (n1/n2/n3 ... n1*n2/n3)
*/MOD	AS ABOVE BUT LEAVE REMAINDER BENEATH (n1/n2/n3 .... rem/n1*n2/n3)
MINUS	CHANGE SIGN OF NUMBER (n1 ...... -n1)
ABS	LEAVE ABSOLUTE VALUE OF NUMBER (n1 ..... Absn1)
+-	LEAVE AS n3 THE VALUE OF n1 WITH SIGN OF n2 (n1/n2 ... . n3)
1+	ADD 1 TO TOP STACK ITEM (n1 ..... n2)
2+	ADD 2 TO TOP STACK ITEM (n2 ..... n2)
2*	MULTIPLY BY 2 THE TOP STACK ITEM (n1 ..... n2)

The following examples should serve to demonstrate the action of the above words:-

25 4 MOD . 1 OK   25/4 = 6 REM 1
25 4 /MOD . . 6 1 OK 25/4 = 6 REM 1
25 4 20 */ . 5 OK   (25*4)/20 = 5
25 4 19 */MOD . . 5 5 OK   (25*4)/19 = 5 REM 5
3 MINUS . -3 OK
-3 ABS . 3 OK
45 -4 +- . -45 OK
4 1+ . 5 OK
4 2+ . 6 OK
8 2* . 16 OK

SINGLE PRECISION STACK OPERATOR WORDS
These words act directly on the values stored on the stack and are as follows:-

WORD	DESCRIPTION
DROP	Remove (destroy) the top stack item (n1 .....)
DUP	Duplicate the top stack item (n1 ... n1/n1)
-DUP	Duplicate the top stack item if it is non-zero-otherwise no action (n1 ... n1/n1) or (n1 ... n1)
SWAP	Swap over the two top stack items (n1/n2 .... n2/n1)
OVER	Make a copy of the 2nd item on the stack to the top of the stack (n1/n2 .... n1/n2/n1)
ROT	Rotate the top 3 items on the stack so that the 3rd moves to the top (n1/n2/n3 .... n2/n3/n1)

Some examples of the use of these words are as follows:-

1 2 3 DROP . . . 2 1 0 . ? MSG # 1
22 44 DUP . . . 44 44 22 OK
1 2 -DUP . . . 2 2 1 OK
2 0 -DUP . . . 0 2 0 . ? MSG # 1
241 19 SWAP . . . 241 19 OK
45 99 63 ROT . . . 45 63 99 OK

Additionally, a further 3 words are provided for interaction between the computation stack we have been discussing and the CPU return stack mentioned previously. These are:-

R   Make a copy on the top of the computation stack of the top item on the Return stack. The Return stack remains unaltered.

>R Transfer the top item on the computation stack to the return stack (Pronounced TO R)

R> Transfer the top item on the return stack to the computations stack (Pronounced R FROM)

The last two mentioned words should be used with care, normally as a pair within a definition, as they alter the return stack contents, which is also used for system control. Usually they are used to temporarily store data from the top of the computation stack when access to an item below is required.

LOGICAL OPERATORS

FORTH supports the following logical operators:-

AND	leaves a bit-by-bit logical AND of the 2 top stack items
OR	leaves a bit-by-bit logical OR of the 2 top stack items
XOR	leaves a bit-by-bit logical EXCLUSIVE-OR of the top stack items
TOGGLE	this word performs a bit-by-bit EXCLUSIVE-OR of the low order byte of the top stack item with the byte whose address is 2nd on the stack, the resultant value is then stored at that address.

As it is sometimes difficult to appreciate logical operations - the user may like to try the following in BINARY and then DECIMAL - so type the following:
: BINARY 2 BASE ! ; OK BINARY OK
11011100 01110111 AND .1010100 OK
11011100 01110111 OR . 11111111 OK
11011100 01110111 XOR . 10101011 OK
DECIMAL OK
92 74 AND . 72 OK
92 74 OR . 94 OK
92 74 XOR . 22 OK

One use of the word TOGGLE can be demonstrated in the definition of SMUDGE (already in the dictionary) in Hexadecimal Base.
: SMUDGE LATEST 20 TOGGLE ;

where LATEST puts on the stack the address of the start of the last word defined in FORTH. &20 is put on the stack and TOGGLE XOR's the byte at LATEST with &20.

RELATIONAL OPERATORS

FORTH supports the following relational operators, and unless specifically stated, all act on signed numbers.

0= Leave a true flag if the top stack item is zero, otherwise leave a false flag.

0< Leave a true flag if the top stack item is less than zero (negative) otherwise leave a false flag.

= Leave true if the top 2 stack items are equal otherwise leave false.

> Leave true of the 2nd stack item is greater than the top stack item.

< Leave true if the 2nd stack item is less than the top stack item.

U< as <, but both numbers are treated as unsigned.

MAX Leave the larger of the 2 top stack items on the stack.

MIN Leave the smaller of the 2 top stack items on the stack.

Note that the relational operators ALL corrupt the stack, for instance in the case of = the two top items ARE LOST and replaced by a true or false flag accordingly. If you wish to retain these numbers they should be both duplicated before the test is applied.

Examples

223 47 0= . . <u>0 223 OK</u>
223 0 0= . . <u>1 223 OK</u>
223 47 0< . . <u>0 223 OK</u>
223 -47 0< . . <u>1 223 OK</u>
223 223 = . <u>1 OK</u>
223 47 = . <u>0 OK</u>
223 47 > . <u>1 OK</u>
223 47 < . <u>0 OK</u>
223 47 MAX . <u>223 OK</u>
223 47 MIN . <u>47 OK</u>

DOUBLE-PRECISION ARTHMETIC

Twos complement form is again used and the range of numbers is -2147483648 to +2147483647 inclusive.

A double precision term may be entered by including a decimal point anywhere in the number as follows:-

26.
684.2743

will place 26 and 6842743 on the stack in double-precision form, the position of the decimal point is stored in the user variable DPL, but in no way affects or alters the number on the stack. Its main use is in numeric output formatting.

FORTH supports the following double-precision operators:-

D+ Double-Precision Add
M* Multiply two signed single-precision values to give a signed double-precision product
U* As above but all numbers are unsigned
M/ Divide the double-precision number second on the stack by the single-precision number on the top, when a single precision quotient will be left. All values are signed.
U/ As above leaving the unsigned remainder and unsigned quotient from the unsigned double-dividend and unsigned divisor
M/MOD Leaves a double quotient and remainder from a double-dividend and single divisor
DMINUS Change the sign of the double-precision value on the stack.
DABS Leave the absolute value of the double-precision item on the stack
D+- Make the sign of the D.P. number 2nd on the stack that of the top single-precision item.

Two other stack operators are provided, these are:-

2DROP Remove the D.P. top stack value
2DUP Duplicate the D.P. top stack value
These two words can also operate on single-precision items, i.e.
2DROP is equivalent to DROP DROP and:-
2DUP the same as OVER OVER

DEFINING NEW FORTH WORDS

As has been remarked earlier, one of FORTH's attributes is its extensibility, and there are several methods of defining new words in FORTH, as well as the colon definitions.

The most commonly used however, are as follows:-

: (COLON definition)
CREATE
CONSTANT
VARIABLE
USER
VOCABULARY

The form which the definition takes does of course vary, depending on which of the above defining words is used, but they all have the same general form, a name 'header' and a parameter 'body'.

The 'header' contains the ascii codes of the letters in the name of the word,

a link address to the previous word in the dictionary and a code-field pointer address, which points to some executable machine-code.

The 'body' of the word can be a number of things, machine-code primitive, lists of addresses (threads) tables of values etc. At the end of this body is a word terminator, either a thread code to machine-code which brings about execution of the next word, or a machine-code instruction which directly transfers control to the next word. In either case a jump through NEXT is executed at the end of the word. How this is achieved will be explained later.

COLON definitions

As has been seen earlier, colon definitions take the form
: NAME ..... ;
where name is followed by a sequence of tasks and the word termination character [;] is used to end the definition.

As we have seen, once defined (and accepted by FORTH!) the word can be typed at the keyboard (EXECUTION MODE) or compiled into a further definition (COMPILATION MODE). Also as has been previously demonstrated, FORTH has the enviable power to FORGET words.

A start to a series of words or applications can be provided with a null, i.e. a word that has no action, the common word for this use in FORTH is TASK, so defining TASK as:-
: TASK ; creates the above mentioned null in the dictionary, any words typed after TASK can be conveniently (if necessary) FORGOTTEN by typing FORGET TASK allowing further (possible more correct?) definitions to be inserted.

CREATE

This defining word is used in the form
CREATE NAME

The header contains the ascii codes of NAME, link field pointer as in [:] and the code field pointer of this word points to its OWN PARAMETER FIELD, or body, which may be filled with executable machine-code. The word CREATE itself is used by [:], CONSTANT etc., and its main use in FORTH is the creation of new machine-code primitive routines, without the need for a Mnemonic Assembler.

Only the header is supplied by FORTH, the body must be hand-compiled using two other words to achieve this. There are respectively [,] and [C,]. [,] takes the two-byte value from the top of the stack and stores it in the first two unused bytes of the parameter body of NAME, incrementing the dictionary pointed by 2, so as to be ready for further byte storage. [C,] takes the low byte value from the top of the stack and stores it in the first unused byte in NAME, again incrementing the dictionary pointer, but this time by only 1.

It should be noted at this point that [,] stores a two-byte value in typical 6502 fashion i.e. low byte first. As an example consider the HEX value FFE6 which we will assume is on the top of the stack. [,] will store this in memory as:-

Dictionary addr n E6
Dictionary addr n+1 FF

It should also be noted that in FORTH leading zeros can be omitted, this can save some typing. An example of the use of CREATE is given below, the word is already in the FORTH dictionary and when the compilation is complete, FORTH will issue error message 4, warning that the word already exists, this will not, however affect its operation.

HEX CREATE XOR
B5 , 255 , 48 C , IB5 , 355 , E8E8 , 4C C , 8242 , SMUDGE

To make things a little clearer, below are the mnemonic and also the 'normal' machine-code formats.

B500	LDA 0,X	Exclusive-or low bytes
5502	EOR 2,X	or top 2 stack items
48	PHA	
B501	LDA 1,X	Exclusive-or high bytes
5503	EOR 3,X	of top 2 stack items
E8	INX	Make room on
E8	INX	stack
4C4282	JMP PUT	Put result on stack

HEX sets the current base to hexadecimal, so that the machine-code may be entered. [.] and [C,] store the appropriate bytes in the parameter field of XOR, note leading zeroes are omitted (if you wish) and SMUDGE is used to smudge the word to be formed in a dictionary search by FORTH, since the use of CREATE smudges the definition to start with. The idea of this is that an incomplete definition will remain SMUDGED i.e. unable to be formed or used. This is part of the compiler security of FORTH. The jump to PUT simply puts the completed XOR'ed bytes on the stack then executes NEXT.

After entering the word it can be tested, its action is identical with that of the XOR already on the nucleus dictionary.

CONSTANT

Numerical values can of course be compiled into a colon definition as literal values, they can also be defined themselves as constants, in the form
n CONSTANT NAME
where n is the value of the constant, CONSTANT the defining word and NAME, the name you wish to give the constant n.

20 CONSTANT SIZE will create a dictionary entry called SIZE and when SIZE is executed it will place the value 20 on the stack, and typing SIZE . will give <u>20 OK</u>

The advantages of using CONSTANT, rather than compiling literal values into a definition are that,

1. If the constant is to be used many times, a saving in dictionary space will be made, despite the initial entry of SIZE, the reason being that literal values require 4 bytes within a definition, whereas a constant requires only 2.

2. If it is desired later to change value, it is far easier to change the value of the constant than having to re-define all of the words which use it. To accomplish this the FORTH words ['] and [!] are used. Executing:- ' SIZE would leave the parameter field address of SIZE on the stack. The use of ! expects 2 values on the stack, a numeric value and an address. It will then store the numeric value at the two bytes starting at this address. Consider:-
23 ' SIZE !

This will give a new value of 23 to the constant SIZE. Typing SIZE . will now give <u>23 OK</u>

VARIABLE

Used in the form
n VARIABLE NAME

Similar to CONSTANT, the above will create a variable with the name NAME and an initial value n.
The difference between CONSTANT and VARIABLE, however, is that an execution of the variable NAME, the ADDRESS at which the numeric value is stored will be placed on the stack.

The operator [@] is then used to retrieve the numeric value, and operator [!] to change it.
Consider the following:-

56 VARIABLE OLDER

This will create a variable OLDER with an initial value of 56, to place this value on the stack, type:-

OLDER @

and of course typing . will print out the value itself.

OLDER @ . <u>56 OK</u>

To change the value of OLDER the following sequence is used.

68 OLDER ! OK

and then typing OLDER @ . gives 68 OK (even older!). Additionally the operator [+!] can be used to increment the value of the variable. Typing 1 OLDER +! will increment the value of OLDER by 1, this increment may be positive or negative, within the single-precision range.

To save typing @ . to print out the value, FORTH provides the operator [?] . its definition is simply:-

: ? @ . ;

and its operation is self explanatory.

USER

This word is used mainly for system modification and its use will be described later.

VOCABULARY

Used in the form:-

VOCABULARLY NAME IMMEDIATE

This version of FORTH has at present three distinct vocabularies FORTH, EDITOR AND GRAPHICS, and the order in which a search will be made, or in which vocabulary a compiled word will be entered, is controlled by the two user variables CONTEXT and CURRENT, each of which contains a pointer to the most recently defined word in a vocabulary.

CONTEXT points to the vocabulary which is to be searched first, and CURRENT points to the vocabulary into which a definition is to be placed. These pointers may be the same or different.

Executing the name of a vocabulary i.e. FORTH, will set CONTEXT to FORTH, the word DEFINITIONS is used to direct to what vocabulary a new definition will be in, by changing the pointer in CURRENT.

i.e. EXECUTING:-

    FORTH DEFINITIONS

will set both CONTEXT and CURRENT to the FORTH vocabulary.

Similarly,

    GRAPHICS DEFINITIONS

will set CONTEXT and CURRENT to the GRAPHICS vocabulary, as a example consider the following (don't forget the spaces).

VOCABULARY PEOPLE IMMEDIATE OK

PEOPLE DEFINITIONS OK

: JIM CR ." THIS IS JIM " ; OK

: FRED CR ." AND THIS IS FRED " ; OK

: JOHN CR ." WHILST THIS IS JOHN " CR ; OK

There is now a vocabulary called PEOPLE, which is as yet not quite over-populated, trying typing JIM FRED JOHN

Now try FORTH JIM FRED JOHN

The same response will be elicited as previously. Typing DEFINITIONS and

trying

JIM FRED JOHN

will however given JIM ?MSG # 0

as the word JIM is not a FORTH word, but a PEOPLE word and since DEFINITIONS has changed CURRENT to FORTH the words JIM FRED and JOHN will not be found. Typing PEOPLE will allow once again a census of the population of this vocabulary, but if new names are typed in they will now be in FORTH, not PEOPLE. This may be a little confusing at first, but with use the situation will become clear.

The word PEOPLE, itself is in the FORTH vocabulary as are FORTH, EDITOR and GRAPHICS, and all the vocabularies link back into the FORTH vocabulary. It is possible to define a vocabulary within another vocabulary other than the FORTH vocabulary but this is not recommended as it will make dictionary searches rather confusing.

If you now type FORGET PEOPLE

the response will be

FORGET ?MSG # 24 which is a prompt to you to declare the vocabulary properly, before trying to forget the word PEOPLE. Remember we are in the PEOPLE vocabulary at the moment so type,

FORTH FORGET PEOPLE OK

Now trying to find either JIM FRED or even JOHN will be very difficult - not only have they been forgotten, but the vocabulary in which they resided has disappeared too. Slightly more useful vocabularies than the one described can, of course, be defined. At this point you may like to execute VLIST in all three vocabularies as follows:-

FORTH VLIST - will give all the current FORTH words
EDITOR VLIST - will give the EDITOR words - then the FORTH words

The defining word CASE defines a word which when executed, expects a case number on the stack. It is used mainly to define words that contain a number of DIFFERENT actions, where only the relevant action, determined by the stack value will be taken. This is best explained by example thus:-

First of all the words to be included in the case-word are defined

```
: JIM ." JIM IS CASE 0 " ; OK
: FRED ." FRED IS CASE 1 " ; OK
: PETER ." PETER IS CASE 2 " ; OK
: SANDRA ." SANDRA IS CASE 3 " ; OK
: CASE <BUILDS SMUDGE ]
DOES> SWAP 2* + @ EXECUTE ;
```

Then the definition which will give the case number

CASE NAMES JIM FRED PETER SANDRA ; OK

Now executing:- 0 NAMES will give:-

JIM IS CASE Ø OK

Similarly executing:- 2 NAMES will give:-

PETER IS CASE 2 OK

IMMEDIATE WORDS

Very often it is required to define a word that will execute even during compilation. Examples of this are the vocabulary words FORTH, EDITOR, GRAPHICS and other words such as conditionals such as IF, THEN etc. Such immediate words will ALWAYS execute, whether in execution or compilation, these then, are IMMEDIATE words, and as we have seen with the definition of PEOPLE - a word can be made IMMEDIATE by typing IMMEDIATE at the end of its definition.

Example:-
: RIGHT-NOW CR ." I HAVE EXECUTED " CR ; IMMEDIATE OK
: LATER CR ." SO HAVE I " CR RIGHT-NOW ; OK

Immediately (!) after pressing the CR after the definition of LATER is completed the message:-

I HAVE EXECUTED
OK
will appear.

Executing LATER will give:-

SO HAVE I
OK

Since it may be necessary to make a word IMMEDIATE inside a COLON definition, i.e. make it execute rather than compile, the FORTH words [(left bracket) and] are used. [serves to terminate compilation and enter the execution mode whilst] terminates execution and enters the compilation mode. These words are usually used as a pair. They may also be used to include a headerless words address into a definition.

: NAME WORDS-COMPILED-NORMALLY [WORDS-THAT-ARE-EXECUTED] WORDS-COMPILED-NORMALLY ;
and an example of the inclusion of a headerless word
HEX

: NAME [addr ,] ; where addr is put on the stack and , removes it into the definition body.

Compilation of Immediate words

Conversely it may be necessary to force the compilation of an IMMEDIATE

word. For this the FORTH word [COMPILE] is used to precede the IMMEDIATE word to be compiled.

For instance using the two previous examples:-

: TOGETHER [COMPILE] RIGHT-NOW LATER ; OK

Notice no message this time other than OK when return is pressed. On executing TOGETHER:-

I HAVE EXECUTED

SO HAVE I
OK

will be printed, showing that RIGHT-NOW has been compiled into the definition of TOGETHER, as an ordinary word - not IMMEDIATE. The word [COMPILE], is itself IMMEDIATE and is not compiled. You may, however, compile it with
[COMPILE] [COMPILE] if you wish to!

Compilation of words into other words

COMPILE

Used in the form
: NAME ... COMPILE WORD ... ;

Here both COMPILE and WORD (together with other words in the definition) are compiled as normal into the definition of NAME. However, when NAME is executed it will compile the execution address of WORD into the next 2 free dictionary spaces. Note that WORD itself is not executed during the execution of NAME. Several examples of the use of forced compilation, conditional compilation etc. are in the FORTH glossary, at the end of the manual.

Conditional operators, loops and branches

We have already discussed the relational operators 0=, 0< etc. Now it is time to discuss the use of the flags they leave on the stack.

Conditional IF .... THEN

The operation of this pair is not the same as it is in BASIC, do not be tempted to skip the following, otherwise your tests will not give the intended results!

FORTH uses conditional tests in a more formally logical and structural approach than does BASIC.

IF tests the top value on the stack, therefore, a relational or other test must leave a flag BEFORE IF can test it.

The use of IF .... THEN is demonstrated as follows;

IF (the test was true) DO-THIS .... THEN DO-THIS-ALWAYS
IF (the test was not true) DONT-DO-THIS .... THEN DO-THIS-ALWAYS

Some operators prefer the use of ENDIF instead of THEN, both have exactly the same meaning, but only THEN is included in this model. As an example of the comparison between BASIC and FORTH the following BASIC program tests for 0 and 1, and makes a decision whether the result is male or female.

```
10 INPUT X
20 IF X=1 ; PRINT "FEMALE" GOTO 40
30 PRINT "MALE"
40 END
```

Here is an example of the same test in FORTH

: XTEST IF ." FE" THEN ." MALE" CR ;

XTEST expects a value on the stack so typing:-

0 XTEST <u>MALE</u>	:	The CR in the definition has been
<u>OK</u>	:	included to separate the OK from
1 XTEST <u>FEMALE</u>	:	both MALE and FEMALE - note that no
<u>OK</u>	:	space has been left to the quotes at the end of each
	:	string MALE" and FE"

Of course, most tests will be a little more difficult to arrange than this somewhat trivial example, and another word ELSE may be included thus:-

IF .... ELSE .... THEN

This combination works as follows

<u>IF</u> (the test is true) DO THIS <u>ELSE</u> AND SKIP TO <u>THEN</u> ... THE WORDS AFTER ELSE ARE NOT EXECUTED IF THE TEST IS TRUE

<u>IF</u> (the test is false) SKIP THIS <u>ELSE</u> DO THIS .... <u>THEN</u> DO THIS
The words after IF are not executed, but the words after ELSE are if the test is false.

In either of the above cases the words (if any) after THEN are executed.

: IOU IF ." YOU OWE ME" ELSE ." I OWE YOU" THEN CR ." SAD! " ;

Testing the above definition with 0 IOU and 1 IOU will print out the appropriate message.

The test IF will remove the flag (true or false) from the top of the stack.

IF .... ELSE .... THEN may be nested to any reasonable depth, however, large multiple nestings may lead to confusion - keep your definitions to a reasonable length.

Examples of properly nested structures,

IF.....<u>IF...ELSE...THEN</u>...THEN...
nested-if

IF.....IF...THEN...ELSE...THEN..
nested-if

The above examples used a flag put manually on the stack, in general, IF will usually, (but not always) be prefixed by a relational operator 0=,= etc. which has previously tested items on the stack.

Indefinite LOOPS

Loops can be either definite or indefinite, depending on the application. The structure
BEGIN........AGAIN

is reasonably self-explanatory. The words between BEGIN and AGAIN are executed and then repeated endlessly, whereas the structure
BEGIN........UNTIL

executes once and then the word UNTIL tests a flag on the stack. If the value is true the operation will be terminated. If however it is false, the whole process repeats again from the word after BEGIN.

The BEGIN .... WHILE .... REPEAT structure will terminate on a false flag tested by WHILE. The words between WHILE and REPEAT are ignored and execution moves to the word(s) after REPEAT. If WHILE finds a true flag, however, the words between WHILE and REPEAT will be executed and then the structure will loop back to BEGIN.

Definite Loops

Any loop whose number of repetitions can be pre-determined will use the DO.....LOOP structure. The word DO removes two items from the stack, the start and limit values.

For example:-

: COUNTER DO I . LOOP ;

Executing the following:-

9 0 COUNTER 0 1 2 3 4 5 6 7 8 OK

I makes a copy of the loop index and . prints it out. One thing will be immediately apparent from the above, the counter started at 0 but ended on 8. The reason for this is as follows:-
The loop index is incremented after the body of the loop is executed, and the loop will terminate when this index equals or exceeds the loop limit. Two important facts emerge from this

1. The body of the loop will be executed at least once irrespective of the value of the loop limit.

2. The last execution of the body within the loop will ne executed with a value I, of 1 less than the limit. Also the word I should only be used inside

the loop, and with care, as it places a value on the stack everytime it executes. In the above example, we have no problem as . then removes it from the stack. If reasonable care is not taken, however, stack overflows can result in system crashes. Remember that the DO.....LOOP structure expects two values on the stack, how they get there is not pertinent, they could be left purposefully from previous calculations or entered from the keyboard.

+LOOP

If it is required to increment a loop by any other value than 1 the DO.....+LOOP structure is provided. The word +LOOP expects a value on the stack and this is automatically added to the LOOP index, for instance:-

```
: 4-COUNTER DO I . 4 +LOOP ;
32 0 4-COUNTER 0 4 8 12 16 20 24 28 OK
```

Both the above examples can be made easier to use with the following

```
: COUNTER 1+ SWAP DO I . LOOP ;
and : 4-COUNTER 1+ SWAP DO I . 4 +LOOP ;
Executing:- 1 7 COUNTER 1 2 3 4 5 6 7 OK
and 0 32 4-COUNTER 0 4 8 12 16 20 24 28 32 OK
```

The pertinence of 1+ and SWAP will be immediately apparent.

The +LOOP structure also allows a negative increment so that the system can be made to count backwards:-

```
: COUNTDOWN DO I . -1 +LOOP ;
0 10 COUNTDOWN 10 9 8 7 6 5 4 3 2 1 OK
```

Loops can be nested, again to a reasonable depth, also conditionals such as IF...ELSE...THEN included within the loop. Ensure, however,, that such structure are nested, and do not overlap.

Constructions such as DO....IF....LOOP THEN are definitely out!

One further word LEAVE, when executed will cause an exit from the loop, its action is not effective immediately but only when the next LOOP is encountered.

LOOP INDICES

The WORD I, as we have seen will leave on the stack the LOOP index. Since it is possible to nest DO....LOOP structures, it may be necessary to put on the stack, the loop indices of outer and inner loops.

Since I gives the inner loop index, it is convenient to give the name J to the next outer loop.

J is not in the nucleus dictionary, but its definition is simply:-

```
: J RP@ 7 + @ ;
```

and in the example:-

```
: n n DO ....
          n n DO ...... I . J .
             LOOP
         LOOP ............. ;
```

J leaves the index of the outer loop, I leaves the index of the inner loop. Similarly if a further nesting is required, and its loop index retrieved, K can be defined as follows

: K RP@ 9 + @ ;

generally speaking, further nestings than 3 will not be required, but the ideas can be expanded if necessary, in the next case the number to be added to the return stack pointer (RP@) will be 4 so:-

: L RP@ 13 + @ ;

NUMBER BASES

This implementation at present supports 2 number bases i.e. DECIMAL and HEX. The current base is stored in a user-variable called BASE and it is quite an easy matter to define alternative number bases, by simply changing the number base stored in BASE. For instance suppose we wanted to define OCTAL.

: OCTAL 8 BASE ! ; <u>OK</u>

typing OCTAL would change the number base to octal or in DUO-DECIMAL

: 12BASE 12 BASE ! ; OK

typing 12BASE would then change the number base to duo-decimal. Any base up to BASE 36 can be established in this way. FORTH's default base, after a WARM or COLD start will always be in decimal.

NUMERIC OUTPUT FORMATTING

We have already met several of the FORTH words that provide numeric output of the value on the stack, listed below are all of these with a description of their operation.

WORD	STACK	DESCRIPTION
	(n....)	Print signed number n followed by a space
.R	(n1/n2..)	Print signed number n1 at the right-hand side of a field n2 characters wide with no following space.
D.	(dn1...)	Print the signed double-number dn1 as in [.]
D.R	(dn1/n2..)	Print the signed double-number dn1 to the right-hand side of a field n2 characters wide, with no following space.
U.	(un1...)	Print the unsigned number un1 in the same format of [.]

The following words should only be used between

<# (Set up for numeric conversion)

and #> (Terminate numeric conversion) and leave ready to TYPE, and all act, on a double-precision number on the top of the stack.

- convert one digit

#S - convert the remaining digits

SIGN - insert a minus sign in the converted term

HOLD - insert a specified character in the converted item

The following example will accept a number up to 21474836 representing millimetres and convert it to metres, the expression will work for positive and negative numbers and the amount put on the stack should include a decimal point i.e. be in double-precision form.

```
: METRES
DUP ROT ROT     (Duplicate the most significant part of the number
                including its sign and save it on the stack)
DABS            (Make the number positive)
<#              (Now start the conversion)
# # #           (Convert 3 digits to give millimetres to right of decimal
                point)
46 HOLD         (Now insert decimal point)
#S              (Convert the remaining digits)
SIGN            (Check on sign of number)
77 HOLD         (Put M at start of number)
#>              (Terminate the conversion)
TYPE SPACE ;    (Display the conversion with following space)
```

Enter the millimetric number in double-precision form.
25789. METRES <u>M25.789 OK</u>
-5893456. METRES <u>M-5893.456 OK</u>

It will be clear from the foregoing example that all numeric conversion starts with the least significant bytes of the number, then proceeds to the most significant. The conversion process takes place in the scratchpad area PAD. As the conversion proceeds towards LOWER memory within PAD, the numeric term is in the correct order to be printed by TYPE (i.e. Highbytes first).

CHARACTER OUTPUT

It was already demonstrated with PLUSES, at the very start of this manual, that a character can be printed on the VDU with the word EMIT, in the form

Character-code EMIT
any character may be used including that of the BBC Microcomputer control-codes e.g. : BUZZER 7 EMIT ; OK
BUZZER simply gives a beep.

TEXT STRINGS

The use of [."] and ["] have already been demonstrated. Note the space between the start of the string and [."] is essential

." THIS IS VALID"
whilst a space at the end of the string to the terminating ["] is not, if a space is included, it will be treated as part of the string. The word [."] is a FORTH word and therefore must be preceded and followed by a space whereas the character " merely terminates the string.

```
."THIS STRING IS NOT VALID" (No space between ." and THIS)
." THIS IS VALID"
." SO IS THIS "
```

Strings can be entered and manipulated in FORTH in a similar manner to that in most high level languages.
Enter the following:-

```
FORTH DEFINITIONS DECIMAL
: STRING <BUILDS DUP C, 0 C, ALLOT
  DOES> 1+ ; OK
: $IN HERE C/L 1+ BLANKS 1 WORD HERE
  PAD C/L 1+ CMOVE PAD DUP C@ 1+ ; OK
: $! 2DUP 1 - C@ 1+ > IF CR
." $ TOO LARGE" 2DROP QUIT THEN
   SWAP CMOVE ; OK
:  $@ COUNT ; OK
   32 STRING WORDS OK
```

First of all a demonstration of the use of the above and then a discussion on what the individual words do.

```
$IN THE WAGES OF SIN IS DEATH OK
WORDS $! OK
WORDS $@ TYPE SPACE THE WAGES OF SIN IS DEATH OK
```

If the following words are defined as follows:-

```
: $LEFT SWAP COUNT ROT MIN ; OK
: $RIGHT SWAP COUNT ROT 2DUP > IF
  DUP >R - + R> ELSE DROP THEN ; OK
```

and trying

```
WORDS 10 $LEFT TYPE SPACE THE WAGES OK
WORDS 12 $RIGHT TYPE SPACE SIN IS DEATH OK
```

The routines can be shortened to the form.

```
: LEFT$ WORDS COUNT ROT MIN TYPE SPACE ; OK
```

and

```
: RIGHT$ WORDS COUNT ROT 2DUP > IF DUP
  >R - + R> ELSE DROP THEN TYPE SPACE ; OK
```

Now executing:-

```
10 LEFT$ THE WAGES OK
12 RIGHT$ SIN IS DEATH OK
```

Similary a definition for mid-string extraction can be made, one possible definition is

: MID$ SWAP WORDS + COUNT ROT MIN TYPE SPACE ; OK

Where 10 2 MID$ OF OK prints the two characters 10 characters into the string and 10 10 MID$ OF SIN IS OK

whilst 5 RIGHT$ 10 6 MID$ DEATH OF SIN OK

will concatenate the end of the string to a selected middle portion.
S! has an inbuilt check on string length. After entering the following:-

$IN WHY ARE THE WAGES OF SIN DEATH ? THEY ARE SOUL DESTROYING OK
And trying to enter the:- WORDS $!
$ TOO LARGE will be given because WORDS is built to accept only 32 characters. It could of course be re-defined to accept more, if necessary, as the definition of STRING will allow for up to 255 characters !

Since STRING is now a defining word i.e. it can be used to define other words, other strings can be called by a different name, for example.
48 STRING PHRASE

will create an empty string called PHRASE, which can be manipulated in the same manner as WORDS. You will notice the use of the words <BUILDS and DOES> in the definition of STRING and these are discussed in the next chapter. PAD has also been mentioned and is an area of temporary storage 64 bytes above HERE which is the address of the first free dictionary space above definitions already existing. From a COLD start, the address of HERE is &1220. Having put several definitions in the dictionary you may like to find out how much memory you have used up to now, so type

HEX HERE U. (Addr) OK
and PAD U. (HERE+64BYTES) OK
The word C/L is simply characters per line allowed in the BBC V2 FORTH
Typing DECIMAL then C/L . 64 OK
will show you that on execution C/L places 64 on the stack whereas HERE and PAD place their respective addresses on the stack. BLANKS is simply that - an ascii BLANK or 32 DECIMAL - remember that a space is used as an unconditional delimiter in FORTH, its use throughout the system is therefore extensive.

CMOVE

Memory Manipulation, i.e. bytes from one location to another is usually accomplished in FORTH by the use of CMOVE its action is as follows:-
(from/ to/count...)

To move n bytes (as specified by count) beginning at addr (from), to addr (to). The contents of addr (from) is moved first, and transfer progresses towards high memory.

Note that in using CMOVE the lowest byte is moved first, so that if the destination and source areas overlap - the command may not satisfy the required result, as the source bytes will be quickly corrupted. This propensity

is used to advantage in the word FILL but can be a disadvantage in other areas.

An alternative method of moving memory could be defined as follows:-

```
HEX OK
: <CMOVE 1 - -1 SWAP DO OVER I + C@
  OVER I + C! -1 +LOOP DROP DROP ;
```

Now execute:- 45 44 43 42 41 3000 5 CFILL OK

3000 3001 5 <CMOVE OK will move the above character string to &3001

and reading will show the correct result has been obtained.

However typing:-
3000 3001 5 CMOVE will simply fill the 5 addressed bytes with the first character encountered.

FILL uses this characteristic to advantage, and the definition
: ERASE 0 FILL ;
and : BLANKS BL FILL ; use the aforementioned CMOVE to fill sections of memory with respectively 0 and BL (32).

You may if you wish define an alternative to CMOVE and <CMOVE which will choose automatically which bytes to move first so that corruption does not take place.

```
: CMOVE >R 2DUP R> ROT ROT -
  IF <CMOVE ELSE CMOVE THEN ;
```

and again try 3000 3001 CMOVE , after first correcting these bytes.

<BUILDS and DOES>

In general the use of FORTH falls into 3 main categories and these can be scaled in levels according to their operation.

LEVEL 0	Simply executing a word
LEVEL 1	Use a defining word e.g. CREATE, : etc. to generate a new word
LEVEL 2	Creation of a new defining word

Since each defining word in FORTH is in itself a compiling word, dedicated to defining a particular type of word-structure, if new word-structures are required that are not already provided for in the FORTH compiling words, a new compiling word must be itself created.

Note that the use of CREATE, : etc. extend the FORTH dictionary and the generation of a new defining word extended FORTH's compiling abilities. Two words are especially provided for this purpose and they are always used together, these are <BUILDS and DOES> and are used in the form:-

: COMMUNITY <BUILDS.....DOES>.... ;
Where optional words may follow both <BUILDS and DOES>.
The word COMMUNITY is in the ordinary sense simply a colon-definition, but the inclusion of <BUILDS and DOES> make it act in a very special way.

First of all the words following <BUILDS are concerned solely with generating a new dictionarly entry for a new word defined by community.

Those following DOES> although compiled into the definition of COMMUNITY are only used by the word defined by COMMUNITY, not COMMUNITY itself. Therefore the words after <BUILDS have the effect at compilation time of the new word; and those after DOES> at its execution.

Having defined COMMUNITY, we can use it to produce new words which although their action will not be identical, will ultimately have certain similarities via the words after DOES>

A definition using COMMUNITY would take the form:-

COMMUNITY MEMBER where MEMBER is a new word being defined by COMMUNITY. The definition may expect values on the stack, but this will depend on the definition of COMMUNITY itself.

Several examples of the use of <BUILDS and DOES> are already in the dictionary, one such is that of the defining word VOCABULARY
VOCABULARY NAME IMMEDIATE
VOCABULARY is defined as follows:-

```
: VOCABULARY <BUILDS A081 , CURRENT @ CFA ,
  HERE VOC-LINK @ , VOC-LINK
        ! DOES> 2 + CONTEXT ! ;
```

an example of its action is in the definition of FORTH i.e. VOCABULARY FORTH IMMEDIATE

Looking at the compiled structure of FORTH, we find the following:-

a) The header FORTH and a link-field address to the previous word in the dictionary (in this case VOCABULARY itself)

b) The code-field address DOES>. .

c) The address of the words immediately following DOES> in the body of vocabulary i.e. 2+ CONTEXT !

d) Word A081

e) Word NTOP (i.e. the top word in FORTH)

f) Word VOC-LO (i.e. the last voc-link pointer - in this case zero). Executing FORTH then executes the word DOES> which serves to put the parameter field address of FORTH on the stack to be operated on by the words 2+ CONTEXT !

In this case the Address of word NTOP is placed in CONTEXT. NTOP always contains the LATEST word in the FORTH vocabulary and the access to this, as has been shown previously, is via CONTEXT i.e. executing CONTEXT @ will place the address of NTOP on the stack and a further @ will leave the

latest word in FORTH hence the definition of LATEST is
: CONTEXT @ @ ;

All of the vocabulary-pointer words FORTH, EDITOR and GRAPHICS have the same general structure as they are all members of the VOCABULARY community. The contents of word NTOP in each entry is of course different.

Since it is apparent from the foregoing that the words NTOP and VOC-LO in each of these WORDS will change, as words are added to each vocabulary, the entries FORTH, EDITOR and GRAPHICS are booted to page 4 (&480) on a COLD start, as their operation within ROM would render this system inoperative, i.e. no new words could be added.

The general form of a word in FORTH is similar irrespective of which method has been used to define it. For the purposes of giving a brief explanation of the inner-structure of FORTH we take the example of a colon definition thus:

```
            NAME                        NAME FIELD ADDRESS
HEAD      LINK POINTER TO PREVIOUS NAME
          CODE FIELD POINTER          code field address
                                        proceding upwards in memory
          EXECUTION ADDRESS 1,                  parameter field address
          EXECUTION ADDRESS 2,
PARAMETER
BODY      EXECUTION ADDRESS 3,

          EXECUTION ADDR        n;
          EXECUTION ADDR OF     ;
```

NFA is the term for the name-field-address of a word
LFA is the term for the link-field-address of a word
CFA is the term for the code-field-address of a word
PFA is the term for the parameter-field-address of a word

Their action is as follows

PFA converts the name-field address to the parameter-field address
LFA converts the parameter-field address to the link-field address
CFA converts the parameter-field address to the code-field address
NFA converts the parameter-field address to the name-field address

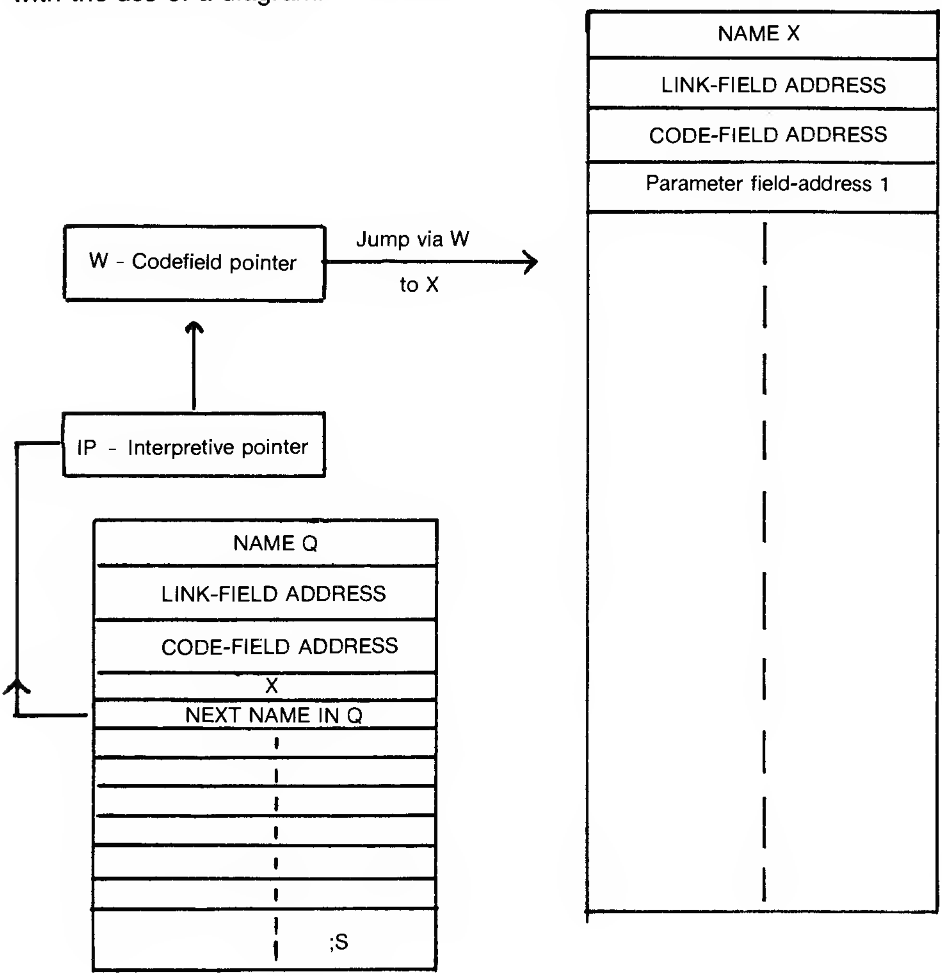
It will be obvious from the foregoing that as each address occupies two-bytes

CFA = LFA +2
PFA = CFA +2

FORTH uses for its operation two main pointers

W - the code-field pointer and
IP - the high-level interpretive pointer

Further, most operations in FORTH are performed by indirection, i.e. an address is loaded into W and an indirect jump to W is taken. The code at the address stored in W is executed and when this is completed the value stored in IP is incremented by 2, the address pointed to be IP is loaded into W and further indirect execution takes place. This is probably explained better with the use of a diagram.



In the diagram IP holds the address next to be executed in WORD Q having previously given the codefield pointer W the address of the codefield for WORD X - WORD X is therefore currently being executed. When this X is executed IP will increment by 2 having transferred to codefield address of (NEXTNAME) to W where the whole process will repeat. At the end of a definition control is returned (via ;S) to the keyboard when the prompt OK will be given.

USER
This word is provided in FORTH for system changes/additions. We have seen that the use of VARIABLE creates a dictionary entry with a parameter body holding the value of the variable. A variable created by USER e.g.:-
HEX 32 USER CHARACTERS
will have as the contents of its parameter body a pointer to the user-area plus &32. Reference to the ROM documentation should be made for the various entries. Newcomers to Forth need not concern themselves at this stage about making use of the user-area, but should study the meaning of the various entries. Of these, the two most important to newcomers are:-
FENCE
A means of protecting entries against the word FORGET e.g.:-
HEX 1356 FENCE !
will move FENCE to &1356, thereby protecting entries below this address.
DP
The dictionary pointer, moving up in value as new definitions are entered. The dictionary can be made to start at a different point to that supplied by COLD e.g.:-
HEX 1600 DP !
will start the dictionary at &1600 instead of &1220.

TAPE INTERFACE and EDITOR

Generally speaking, most FORTH applications or programs, will be entered in the form of "Screens". A Screen usually consists of a series of definitions, and the size of the Screen in terms of memory, is usually arranged to co-incide with the text-display area of the computer is use.
The BBC micro-computer utilises 1K bytes of memory in normal text mode, and so in this implementation of FORTH, a "Screen" is arranged to consist of 16 lines of 64 characters, making a total of 1024 (1K) bytes.
Text is entered onto lines numbered 0-15, and the text thus entered in a screen may be corrected, deleted or finally entered into the FORTH dictionary. This method of application entry should form the main programming tool of the user, as definitions can be entered, tested and modified if necessary, with the minimum of typing.
A separate VOCABULARY is provided in FORTH for the manipulation of the screen editor, and the definitions in this vocabulary are therefore EDITOR DEFINITIONS.
The screen editor is intimately linked to the cassette interface, so that definitions may be saved in high-level form rather than compiled FORTH, although this to, is possible.
The user should familiarise him/herself with all the EDITOR commands, and possibly the best way of doing this is by example, so type:-
PROGRAM this should elicit the response:-
FIRST SCREEN NUMBER ? and since this is your first screen in FORTH type:-
0 The program will respond by typing out a blank "Screen" as follows:-

```
Scr # 0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

The above is to show you that screen 0 is at your disposal and as yet contains no information. In fact the area of memory mapped by the screen editor, &E00 to &1200, has been filled with ascii blanks, or spaces, and is ready to be filled with your applications. Executing PROGRAM automatically enters the EDITOR vocabulary, so we are ready to enter some text, using the EDITOR commands.

P - Put following text on the screen line indicated by the number on the stack. If you type in the following, as always, end an entry with a return, the program will respond in familiar fashion with an OK:-

0 P AN EXAMPLE ONLY

You can now check to see indeed if the text has been entered in the screen by typing:- L

L - List the current screen contents and number, and display the position of the string-editor cursor (more of this particular animal later)

The text you have just entered should be on line 0 of the screen. We can move the entered line down one line with the command:- S

S - Spread the text by inserting a blank line, line 15 is lost. After typing S and then L the text should appear on line 1.

Now type:- 1 TEXT THIS IS

1 TEXT waits for text input and holds this text at PAD, we can insert the PAD text with the command:- I

I - Insert the text from PAD at the line specified on the stack, lower lines are moved down, and line 15 is lost.

So typing:- 0 I and then L will reveal on line 0:- THIS IS and on line 2:- AN EXAMPLE

Now try:- 0 D and again list the screen with L

D - Delete the given line from the screen, and move up all the lines to close the gap, but save the deleted line at PAD.
Now type:-   0 R   and again list the screen

R - Replace the contents of a line with text at PAD
Now type:-   1 H   and list the screen.

H - Hold the text of the given line at PAD. The line also remains in the screen.
Typing:-   2 R   will now put the contents of PAD at line 2
Typing:-   0 E   and listing the screen, will erase line 0

E - Erase the line given on the stack, leaving it blank. The contents of the line are not saved at PAD.
Finally type:- 1 T   and the text of line 1 should be printed out.

T - Type the contents of the given line, and also copy it at PAD, the text also remains in the screen.

The above demonstration although trivial, should serve to familiarise the user with the screen editor commands, and it should be noted that all of the commands which expect following text may be used with the machines' cursor control and copy keys, although care should be taken to insert a space after the command word BEFORE moving the cursor about on the screen, and copying text.

STRING EDITOR
This implementation also provides a string editor, and the cursor for this facility is the # symbol. The commands are summarised as follows:-

TOP - Reset the string cursor to the top of the current screen.

C - Insert the given text at the current cursor position. Use as:-
    C text

F - Find the given text, and position the cursor immediately after its first occurence. Use as:-
    F text

TILL - Delete all text, from the current cursor position to the end of the given text. This command will only work within 1 line. Use as:-
    TILL text

X - Find and delete the first occurence of the given text. Use as:-
    X text
The next commands do not require keyed text, but use the text stored at PAD.

N - Find the next occurence, in the screen, of the text at PAD. Use simply as:-
    N

B - Move the cursor back by the character count of the text at PAD. Again, used simply as:-
    B
The next two commands expect a character count on the stack.

DELETE - Delete by the number on the stack, n characters backwards from

the current cursor position. Use this as:-
n DELETE

M - Move the cursor backwards or forwards by the number on the stack. Use a negative number to move backwards. e.g.:-
-10 M would move the cursor back 10 character positions.
All of the above commands which search for text, use the definition MATCH, a powerful machine-code primitive, whose action is as follows:-

MATCH - Search an area of memory for an occurence of text whose start address and character count is known. The parameters required by MATCH are as follows:-
addr1 bytelength addr2 charcount MATCH where addr1 is the start address of memory to be searched, bytelength is the number of bytes in the search area, addr2 is the start address of the text to be matched (usually that of PAD) and charcount the number of characters in the string. MATCH will leave on the stack the following:-
.....flag offset
The flag, if true (non-zero) indicates a successful match, and is zero if the search is unsuccessful, offset is the number of bytes from addr1 to the end of the matched string, and so for a successful search the start address of the matched string is simply:-
(addr1 + offset - charcount)

LIST, SAVE, LOAD, ENTER, ;S and -->

As has been mentioned earlier, the screen EDITOR and tape interface are intimately linked. Screens are usually given a number between 0 and 999, and it is this number, rather than a name, which is used for identification purposes by the tape interface. Once again the user should be familiar with the relevant commands before attempting any serious programming, and the following definitions, put on a screen using the EDITOR commands, then saved and then re-located, should familiarise the user with the effect of the various commands.
First of all type:- PROGRAM then enter a start screen number e.g. 10. After this, enter the text using the EDITOR commands. The program is that to give the nearest square root of a given number, and is based on a well known algorithm.

```
SCR # 10
(BBC FORTH SQUARE ROOT SOLVER)
FORTH DEFINITIONS DECIMAL
: PICK (Copy the nth item on the stack to the top of the stack)
DUP 1 < 5 ?ERROR DUP + SP@ + @ ;
: ROOT DUP 2 / (Give name to procedure and do first approx.)
BEGIN
   2DUP / + 2 / (Calculate next approximation)
   DUP (Duplicate value for two tests)
   1 - DUP * 3 PICK < (Do first test)
```

9 >R (Save test flag)
10 DUP (Save value on stack for after next test)
11 1+ DUP * 3 PICK > (Do second test)
12 R> (Bring back flag from previous test)
13 AND (Logically AND both flags)
14 UNTIL (Now test remaining flag, if not true go back to BEGIN)
15 SWAP DROP ; ;S (Leave square root on stack)

Note the use of:- ;S at the end of the screen, this serves to terminate interpretation of the screen and return control to the keyboard. If all seems correct, type:-
ENTER and after a brief pause, you should be rewarded with the usual OK.
The definition ROOT can now be tested, e.g.:-
810 ROOT . should give 28 OK and:-
712 ROOT . should give 26 OK whilst:-
9999 ROOT . will give 100 OK

It is quite usual in FORTH to comment a screen in the manner exampled above. The interpreter will simply ignore all text following the FORTH word:- (until the character:-) is reached. Note that:- (, being a FORTH word, requires a space either side of it for correction operation. Errors during screen interpretation, will be trapped and the appropriate message given. As is usual in FORTH a partially completed definition due to error, will be left in the dictionary in a "smudged" form, and this should be removed before re-compiling.

In the event of screen compilation collapse due to an error, the word:- WHERE if typed will list the offending line where the interpreter thinks the error occurred.
It can become very laborious typing SMUDGE FORGET (BAD-DEFINITION) every time, and some FORTH users have resorted to a very simply ploy based on our old friend TASK. TASK as we know is simply a null in the dictionary, and does nothing - except save considerable typing when errors and/or changes in definitions necessitate re-compilation. The method is simply as follows. Define the word TASK before beginning to compile from your screen, e.g.:-
: TASK ; and then lay out the screen thus:-

SCR # 10
0 (Screen Title)
1 FORTH DEFINITIONS FORGET TASK : TASK ;
2 Start of program

Now every time you type ENTER the previous TASK is forgotten, along with all those defitions entered after TASK. TASK is again entered, at the same point in the dictionary, and your new, beautifully corrected, definitions are compiled again. This very simply tool saves all the bother of typing SMUDGE

and FORGET, over and over again.
To return to our little program. You may wish to add the above simple tool, using the EDITOR commands, then save it on tape.
First of all, decide on which Baud rate your machine will handle. The default rate (after a BREAK) is 1200 Baud, and is unsuitable for some cassette machines. If this is the case, type:-

MONITOR TAPE 3 then type:-

SAVE the response to this command will be:-

>10 RECORD then RETURN So do this. The normal tape routine parameters will be printed out, and when your program is saved, the response OK will be printed.

Now type:- 0 SCR ! 10 LIST to which the response will be:-

>10 Searching and if you rewind the cassette and play it, the program will be reloaded in to the screen buffers and then displayed.

Now type:- EMPTY-BUFFERS and then:-

0 SCR ! 10 LOAD again the response:-

>10 Searching will be given. This time, when the file is found, it will be automatically ENTERed, and if the previous definitions were not FORGOTen by you, the standard warning message 4 will be elicited.

To summarise the above commands:-

LIST - List the screen whose number is on the stack, and if this is the current screen number, LIST this, otherwise prompt for a file elsewhere (e,g, cassette). Use as:-

n LIST

SAVE - Save the current screen to file, the filename being the screen number. Use as:-

SAVE

LOAD - Load the numbered screen into the screen buffers, and then ENTER it (i.e. compile into dictionary). Use as:-

n LOAD

;S - Terminate interpretation of a screen, returning control to the keyboard.

Further words we have not yet discussed are:-

--> - Continue interpretation on the next screen, automatically incrementing the screen number, and searching the filing system for this numbered screen. Very often a program will not fit onto one screen, even if bracketed comments are left out. This word is used at the end of the screen instead of:- ;S , the system will then automatically look for and then ENTER the next screen. For long applications, over several screens this word is used at the end of every one except the last, when again ;S should be used.

MORE - Save present screen to filing system, and when this is done, increment the screen number and clear the buffer area, ready for the next part of the program entry.
You may wish to escape from any of the above tape routines, and this is done

by simply pressing the ESCAPE key. The system will return control to you via a WARM start, with all your applications intact.

DISCS

At the time of writing, the Disc Filing System for the BBC Microcomputer has not yet been released. However, this implementation is fully compatible with Disc use, and no problems should be encountered. Disc users shsould refer to their operating manuals, and also to the section in this manual on memory allocations.

GLOSSARY OF THE FORTH NUCLEUS DICTIONARY

In this glossary the following criteria will apply. The first line of each entry gives a symbolic description of the action by the procedure on the parameter stack. These symbols indicate the order of entry onto the stack. The dotted "....", indicates execution of the procedure, and, any parameters left on the stack are then listed. In all cases as is common practice, the top of the stack is to the right of any parameter list.
The symbols used in this glossary are given below, together with their intended meanings.

addr	memory address
b	8 bit byte (i.e. higher 8 bits zero)
c	7 bit ascii character (higher 9 bits zero)
d	32 bit signed double integer, the most significant portion with sign on the top of the stack
f	boolean flag. 0=false, non-zero=true
ff	boolean false flag=0
n	16 bit signed integer number
u	16 bit unsigned integer number
tf	boolean true flag=non-zero

The capital letters on the right will denote definition constraints or other characteristic.

C	may only be used with a colon definition. A digit indicates no. of memory addresses used if more than one.
E	intended for execution only
P	has the precedence bit set. Will execute even when compiling
U	a user-variable

Unless contra-indicated, all numbers are 16 bit signed integers, the high byte is on the top of the stack, with the sign in the leftmost bit. For 32 bit signed double numbers, the most significant part, including the sign, is on top. All arithmetic is 16 bit signed integer, error and underflow indication is not specified.

! n addr ....
Store 16 bit n at address. Pronounced "store"

!CSP Save the stack position in CSP.
Part of the compiler security system.

d1....d2
Generate from double number d1, the next ascii character, to place in an output string. Used between <= and =>

#> d....addr count

Terminate numeric output conversion by dropping d, leaving the text address and char. count ready for TYPE

#S d1....d2

Generate ascii text in the output buffer, by the use of #, until a zero double number results. Used between <# and #>.

' ....addr P

Used in the form:-

' nnnn

Leaves on the stack, the parameter field address of the word nnnn. Executes in a colon definition to compile the address as a literal value. Pronounced "tick".

(P

Used in the form:-

(cccc)

Ignore a comment delimited by a following right parenthesis on the same line. A space is required after the leading parenthesis, an optional sapce may be used before the delimiter, if required.

(.") C+

The run-time procedure, compiled by ." which transmits the following in-line text to the output device. See ."

(;CODE) C

The run-time procedure, compiled by ;CODE, that re-writes the code field of the most recently defined word to point to the following machine-code. See ;CODE

(+LOOP) n.... C2

The run-time procedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(ABORT)

Executes after an error when warning is -1. This word normally executes ABORT, but may be altered by the user, to an alternative procedure. This change should be carried out with care.

(DO) C

The run-time procedure compiled by DO, which moves the loop control parameters to the return stack. See DO.

(FIND) addr1 addr2...pfa b tf (ok)

addr1 addr2...ff (bad)

Searches the dictionary starting at the name field address addr2, matching to the text at addr1. Returns the parameter field address, length byte of name field and boolean true if found. If no match is found, only a false flag is left.

(LOOP)   C2
The run-time procedure compiled by LOOP, which increments the loop index and tests for completion. See LOOP

(NUMBER)   d1 addr1....d2 addr2
Convert the ascii text beginning at addr1+1 with regard to current BASE. The new value is accumulated into double d1, and left as d2. addr2 is the address of the first unconvertable digit. Used by NUMBER.

*   n1 n2....product
Leave the signed product of two signed numbers.

*/   n1 n2 n3....n4
Leave the ratio n4=n1*n2/n3 where all are signed numbers. Retention of an intermediate 31 bit product gives greater accuracy than that of the sequence:-

n1 n2 * n3 /

*/MOD   n1 n2 n3....n4 n5
Leave the quotient and remainder of the operation:-

n1*n2/n3

Again, a 31 bit intermediate product is used, as for */.

+   n1 n2....sum
Leave the sum of n1+n2.

+!   n addr....
Add n to the contents of addr. Pronounced "plus-store".

+-   n1 n2....n3
Apply the sign of n2 to n1, leaving this as n3

+LOOP   n1....(run)
         addr n2....(compile)   P,C2
Used in the colon-definition in the form:-

DO .... n1 +LOOP

At run time, +LOOP controls branching back to a corresponding DO. This branch is controlled by n1 and the loop limit. The signed n1 is added to the loop index and the total compared to the loop limit, a branch occuring until the new index is equal to or greater than the loop limit (n1>0) or the new index is equal to or less than the loop limit (n1<0). On completion of the loop, all loop control parameters are discarded, and execution moves to whatever follows +LOOP.

+ORIGIN   n....addr
Leave the memory address relative by n to the origin parameter area. Used to access the origin boot-up parameters. Note in this implementation these parameters can not be modified, the user can of course alter these in the user area (&400), although care should be exercised if this done. to avoid a system crash.

,   n....
Store n into the next available dictionary memory cell, advancing the

dictionary pointer. Pronounced "comma".

-               n1 n2....diff
Leave the difference of n1-n2.

-->   P
Continue interpretation on next screen. Pronounced "next-screen".

-DUP   n1....n1 (if zero)
                n1....n1 n1 (non-zero)
Reproduce n1 if it is non-zero, otherwise do nothing. Usually used to copy a value just before IF, eliminating the need for an ELSE part to drop it.

-FIND   .... pfa b tf   (found)
                .... ff   (not found)
Accepts the next text word (delimited by blanks) and searches the CONTEXT and CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true flag are left on the stack. Otherwise only a boolean false flag is left.

-TRAILING   addr n1....addr n2
Adjusts the character count n1 of a text string beginning address to suppress the output of trailing blanks. That is, the characters at addr+n1 to addr+n2 are blanks.

.   n....
Print a number from a signed 16 bit two's complement value, converted according to the numeric BASE. A trailing blank follows. Pronounced "dot".

."   P
                Used in the form:-
                ." cccc"
Compiles an in line string cccc (delimited by the trailing ") with an execution procedure to transmit the text to the selected output device. If executed outside a definition, ." will immediately print the text until the final ".

.LINE   line scr....
Print on the output device, a line of text from the line number and screen number on the stack. Trailing blanks are suppressed.

.R   n1 n2....
Print the number n1 right aligned in a field whose width is n2. No following blanks are printed.

/   n1 n2....quot
Leave the signed quotient of n1/n2.

/MOD   n1 n2....rem quot
Leave the remainder and signed quotient of n1/n2. The remainder has the sign of the dividend.

0 1 2 3 ....n

These small numbers are used so often that they have been defined by name in the dictionary as constants. Their use in this way saves considerable space within definitions, as a constant will occupy only 2 bytes of memory, whereas a literal value will occupy 4.

0< n....f

Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0= n....f

Leave a true flag if the number is equal to zero, otherwise leave a false flag.

0BRANCH f....

The run-time procedure to conditionally branch. If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by:- IF, UNTIL and WHILE.

1+ n1....n2

Increment n1 by 1, and leave as n2.

2+ n1....n2

Increment n1 by 2, and leave as n2.

: P,E

Used in the form called a colon definition:-

: cccc .... ;

Creates a dictionary entry defining cccc as equivalent to the following sequence of FORTH word definitions:-

'....' until the next:- ; or:- ;CODE.

The compiling process is done by the text interpreter as long as STATE is non-zero. The CONTEXT vocabulary is set to CURRENT, and the words with the precedence bit set (P) are executed rather than compiled.

; P,C

Terminate a colon-definition and stop further compilation. Compiles the run-time:- ;S.

;CODE P,C

Used in the form:-

: cccc ... ;CODE (following machine-code.)

Stop compilation and terminate a new defining word cccc by compiling:- (;CODE). When cccc later executes in the form:-

cccc nnnn

The word nnnn will be created with its execution procedure given by the machine-code following cccc. That is, when nnnn is executed, it does so by jumping to the code after nnnn. An existing defining word must exist in cccc prior to:- ;CODE.

;S P

Stop interpretation of a screen. ;S is also the run-time word compiled at the end of a colon-definition which returns execution to the calling procedure.

< n1 n2....f

Leave a true flag if n1 is less than n2, otherwise leave a false flag.

<#

Set up for numeric output to terminal output device using the words:- <# # #S SIGN #>. The conversion is done on a double number producing text at PAD.

<BUILDS

Used within a colon-definition in the form:-

: cccc <BUILDS .... DOES> .... ;

Each time cccc is executed, <BUILDS defines a new word with a high-level execution procedure. Executing cccc in the form:-

cccc nnnn

uses <BUILDS to create a dictionary entry for nnnn with a call to the DOES> part of nnnn. When nnnn is later executed, it has the address of its own parameter area on the stack, and executes the words after DOES> in cccc.

= n1 n2 .... f

Leave true if n1=n2, otherwise leave false.

> n1 n2 .... f

Leave true if n1 is greater than n2, otherwise leave false.

>R n ....

Remove the top stack item and place it as the most accessible item on the return stack. Use should be balanced with the use of R> within the same definition.

? addr ....

Print the value contained at the address in free format according to the current BASE.

?COMP

Issue an error message if not compiling.

?CSP

Issue an error message if stack position differs from value saved in CSP.

?ERROR f n ....

Issue an error message number n, if the boolean flag f is true.

?EXEC

Issue an error message if not executing.

?LOADING

Issue an error message if not loading.

?PAIRS n1 n2 ....
Issue an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK
Issue an error message if the stack is out of bounds. i.e. The action of a word will either move the stack pointer past its upper or lower limits.

?ESC(?TERMINAL) .... f
Leave true if the ESC key has been pressed. The ESC key should only be pressed if executing a procedure which contains ?ESC. To abort execution from other procedures in this implementation, the BREAK key should be used.

@ addr .... n
Leave the 16 bit contents of address as n.

ABORT
Clear the stacks and enter the execution state, returning control to the operator after printing the installation identity.

ABS n .... u
Leave the absolute value of n as u.

AGAIN addr n .... (compiling) P,C2
Used in a colon-definition in the form:-

BEGIN .... AGAIN

At run-time, AGAIN forces execution to return to corresponding BEGIN. There is no effect on the stack, and execution cannot leave this loop, unless R> DROP is executed one level below, or the BREAK key is pressed. At compile time, AGAIN compiles BRANCH with an offset from HERE to addr. n is used for compile time error checking via ?PAIRS.

ALLOT n ....
Add the signed number n, to the dictionary pointer DP. May be used to reserve space or to re-origin the users dictionary starting point.

AND n1 n2 .... n3
Leave the bit-wise logical AND of n1 and n2 as n3.

BACK addr ....
Calculate the backward branch offset from HERE to addr, and compile into the next available dictionary address space.

BASE .... addr U
A user variable containing the current number base for numeric output.

BEGIN .... addr n (compiling) P
Used in a colon-definition in the forms:-

BEGIN .... AGAIN
BEGIN .... UNTIL
BEGIN .... WHILE .... REPEAT

At run-time, marks the starting point for a repetitive sequence to be executed. At compile time BEGIN leaves its return address and n for compiler error checking.

BL c
A constant which leaves the ascii value for a blank.

BLANKS addr count ....
Fill an area of memory with ascii blanks, starting at addr for count bytes.

BLK .... addr U
A user variable, which if zero forces input interpretation from the terminal keyboard, otherwise interpretation is from the text "screen"

BRANCH C2
The run-time procedure to unconditionally branch. An in-line offset is added to the interpretive pointer IP to branch ahead or back. BRANCH is compiled by:- ELSE, AGAIN and REPEAT.

C! b addr ....
Store single byte at address.

C, b ....
Store single byte b into next available space in dictionary, incrementing the dictionary pointer DP by 1.

C addr .... b
Leave on the stack the 8 bit contents of address.

CFA pfa .... cfa
Convert the parameter field address of a definition to its code field address.

CMOVE from to count ....
Move the specified quantity of bytes from address to address, moving the lower addressed bytes first.

COLD
The cold start procedure to adjust the dictionary pointer to its implementation starting point, then restart the system via ABORT. All previous users applications are destroyed, ready for the insertion of new applications.

COMPILE C2
When the word containing COMPILE executes, the execution address of the word following COMPILE is compiled into the dictionary. This allows for specific compilation situations to be handled in addition to the normal compilation as done by the interpreter.

CONSTANT .... addr U
This is a defining word and is used in the form:-

n CONSTANT cccc

to create a word cccc, with its parameter field containing n. When cccc is executed, it will push the value n onto the stack.

CONTEXT .... addr U

The user-variable containing a pointer to the vocabulary within which dictionary searches will be first made.

COUNT addr1 .... addr2 n

Leave the byte address addr2 and byte count n of a text message beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text begins with the second byte.

CR

Transmit a carriage-return/line-feed to the output device.

CREATE

A defining word used in the form:-

CREATE cccc

by such words as CONSTANT, to create a header for a definition. The code field contains the address of the words own parameter field. The new word will be created in the CURRENT vocabulary. Most often used in FORTH to compile machine-code into the dictionary, where a mnemonic assembler is not available.

CSP .... addr U

A user variable storing the stack pointer position, for compilation error checking.

D+ d1 d2 .... dsum

Leave the double-number sum of two double-numbers.

D+- d1 n .... da

Apply the sign of n to the double-number d1, and leave as d2.

D. d ....

Print a signed double number from a 32 bit two's complement value. The high order 16 bits are most accessible on the stack. Conversion is performed in the current numeric BASE, and a blank follows.

D.R d n ....

Print a signed double number d right aligned in a field n characters wide.

DABS d .... ud

Leave the absolute value ud of a double number.

DECIMAL

Set the numeric conversion BASE to decimal input/output.

DEFINITIONS

Used in the form:-

cccc DEFINITIONS

Set the CURRENT vocabulary to the CONTEXT vocabulary. In the example executing vocabulary name cccc made it the CONTEXT vocabulary, and executing DEFINITIONS made both specify vocabulary cccc.

DIGIT c n1 .... n2 tf (ok)
c n1 .... ff (bad)
Converts the ascii character c (using base n1) to its binary equivalent n2, together with a true flag. If the conversion is invalid, only a false flag is left.

DLITERAL d .... d (executing)
d .... (compiling) P
If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it onto thestack. If executing, the number will remain on the stack.

DMINUS d1 .... d2
Convert d1 to its double number two's complement d2.

DO n1 n2 .... (execute)
addr n2 .... (compile) P,C2
Occurs in a colon definition in the form:-
DO .... LOOP
DO .... +LOOP
At run-time, DO begins a sequence with control of repetition controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. Upon reaching LOOP the index is incremented by one, and until the index either equals or exceeds the limit, execution of the loop is repeated.

DOES>
A word which defines the run-time action within a high-level defining word. DOES> alters the code field and first parameter of the new word to execute the sequence of compiled word addresses following DOES>. Used in combination with <BUILDS. When the DOES> part executes, it begins with the address of the first parameter of the new word on the stack. This allows interpretation using this area or its contents. Typical uses include the FORTH assembler, multi-dimensional arrays, and compiler generation.

DP .... addr U
A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT, or by the sequence:-
addr DP !
when the next dictionary entry will be located from addr.

DPL .... addr U
The user variable containing the number of digits to the right of the decimal point in a double number input.

DROP n ....
Drop the top stack item.

DUP n .... n n
Duplicate the top stack item.

ELSE addr1 n1 .... addr2 n2 (compiling) P,C2
Occurs within a colon definition in the form:-

IF .... ELSE .... THEN

At run-time, ELSE executes after the true part following IF. ELSE forces execution to skip over the following false part and resumes execution after THEN. At compile-time the necessary offsets are compiled by BRANCH, to carry out the above operation.

EMIT c ....
Transmit ascii character to the output device. OUT is incremented for each character output.

EMPTY BUFFERS
Clear the tape/screen editor area by filling it with ascii blanks.

ENCLOSE addr1 c addr1 n1 n2 n3
The text-scanning primitive used by WORD. From the text address addr1 and an ascii delimiting character c, is determined the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included. This procedure will not process past an ascii null, or blank, treating this as an unconditional delimiter.

ERASE addr n ....
Clear a region of memory to zero from addr over n addresses.

ERROR line .... in blk
Execute error notification and restart of system. If WARNING is zero, n is printed as an error message number in non-disc systems. If WARNING is -1, the definition (ABORT) is executed, which usually executes the system ABORT. The user can however cautiously modify this event by altering (ABORT). The contents of IN and BLK are used to assist in determing the location of the error. The final action is via QUIT.

EXECUTE addr ....
Execute the definition whose code field address is on the stack.

EXPECT addr count ....
Transfer characters from terminal to addr, until a “return”, or the allowable character count is reached. One or more nulls are added to the end of the text.

FENCE .... addr U
The user variable containing the pointer below which definitions may not be forgotten. FENCE may be modified thus:-

addr FENCE !

FILL addr quan b ....
Fill memory at the address with the specified quantity of bytes b.

FIRST .... addr
A constant leaving the address of the first tape/screen buffer.

FORGET E
Executed in the form:-

FORGET cccc

deletes the definition cccc from the dictionary together with all entries following it in memory. An error message will be given if the CONTEXT and CURRENT vocabularies are not one and the same on execution of FORGET.

FORTH P
This is the name of the primary vocabulary, execution will make FORTH the CONTEXT vocabulary. As the word is immediate it will execute during the creation of a definition, so that the vocabulary can be selected at compile-time. All other vocabularies ultimately link to FORTH, and the user should beware of defining vocabularies which do not, as this will make dictionary searches difficult to achieve.

HERE .... addr
Leave the address of the next available dictionary location.

HEX
Set the numeric conversion BASE to sixteen (hexadecimal).

HLD .... addr
A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD c ....
Used between <# and #> to insert an ascii character into a pictured numeric output string, e.g.:-

2E HOLD

will place a decimal point.

I .... n
Used within a DO-LOOP to copy the loop index to the stack.

ID. addr ....
Print a definitions name by its name field address.

IF f .... (run-time)
.... addr n (compile) P,C2
Occurs in a colon definition in the form:-

IF (tp) .... THEN
IF (tp) .... ELSE (fp) .... THEN

At run time, IF selects execution based on a boolean flag. If f is true (non-zero), execution continues ahead through the true part of the definition, whereas if the flag (zero), execution skips to the false part after ELSE. After either part is executed, the part (if any) after THEN is executed. ELSE and its false part are, of course optional, and if omitted, false execution skips to the THEN part.
At compile-time IF compiles OBRANCH and an offset to carry out the above procedure. addr and n are used for resolution of the offset and error testing.

IMMEDIATE
Mark the most recent dictionary entry so that when encountered at compile-time, it will execute rather than be compiled. This is accomplished by setting the precedence bit in the name header. This method allows definitions to handle unusual compiling situations, rather than build then into the fundamental compiler. The user can force compilation of an IMMEDIATE word by preceding it with [COMPILE].

IN .... addr
A user variable containing the byte offset within the current input text buffer (keyboard or screen), from which the next text will be accepted. WORD uses and moves the value of IN.

INTERPRET
The outer text interpreter which sequentially executes or compiles text from the input stream (terminal or screen), depending on STATE. If the word name cannot be found after a search of CONTEXT and then CURRENT vocabularies, it is converted to a number according to the current BASE. This also failing, an error message is given together with the offending text. Text input will be taken according to the convention for WORD. If a decimal point is found as part of a number, a double-number value will be left. The decimal point serves no other purpose other than to accomplish this action. See NUMBER.

KEY c
Leave the ascii value of the next key struck.

LATEST .... addr
Leave the name field address of the last dictionary entry in the CURRENT vocabulary.

LEAVE C
Force termination of a DO-LOOP at the next opportunity, by setting the loop limit to equal that of the current index. The index remains unaffected, and execution will continue as normal until either LOOP or +LOOP is encountered.

LFA pfa .... lfa
Convert the parameter field address of a dictionary entry to its link field address.

LIMIT .... addr
A constant leaving the address just above the screen/tape-interface buffers.

LIST n ....
Display the ascii text of the screen n. If the value of SCR is equal to n, then the current contents of the screen buffers will be displayed, otherwise the value of SCR will be changed to n, and a search of the filing system made for the numbered screen n.

LIT C2

.... n

Within a colon-definition, LIT is automatically compiled before each 16 bit number encountered in input text. Later execution causes the in line literal value to be pushed onto the stack.

LITERAL P,C2

n .... (compiling)

If compiling, then compile the 16 but value n as a literal value. The definition is IMMEDIATE so will act during compilation in a colon-definition. Intended use is as follows:-

: xxxx [calculate] LITERAL ;

Compilation is suspended for the compile-time calculation of a value, the compilation is resumed and LITERAL compiles this value.

LOAD n ....

Begin interpretation of screen n. Loading will terminate upon encountering ;S. See ;S and -->.

LOOP P,C2

addr n .... (compiling)

Occures in a colon-definition in the form:-

DO .... LOOP

At run-time, LOOP selectively controls branching back to a corresponding DO, based on the loop index and limit. The loop index is incremented by one and compared with the loop limit, a branch back to DO occuring until the index equals or exceeds the limit, at this time the loop control parameters are discarded, and execution carries on with the next part of the definition, At compile-time LOOP compiles (LOOP) to carry out the above procedure, using addr to calculate the offset and n for error trapping.

M* n1 n2 .... d1

Leave the double-number signed product of the signed numbers n2 and n2.

M/ d n1 .... n2 n3

Leave the signed remainder n2 and signed quotient n3, from the double number dividend d and divisor n1. The remainder takes its sign from the dividend.

M/MOD ud1 u2 .... u3 ud4

Leave a double quotient ud4 and remainder u3 from a double dividend ud1 and single divisor u2.

MAX n1 n2 ... max

Leave the greater of the two numbers.

MESSAGE n ....

Print on the selected output device, the message number n as:-

MSG # n

The procedure may be diverted by the user to print out an error message in

full, rather than the number by itself. See the section on ERRORS and (ABORT).

MIN n1 n2 .... min
Leave the smaller of two numbers.

MINUS n1 .... n2
Leave the twos' complement of a number.

MOD n1 n2 .... mod
Leave the remainder of n1/n2, with the sign of n1.

MONITOR
Exit from FORTH to an operating system call saving the re-entry point. In this implementation MONITOR is used as follows:-

MONITOR command

where command may be any of the commands normally preceded by a *.

NFA pfa .... nfa
Convert the parameter field address of a definition to its name field address.

NUMBER addr .... d
Convert a character string left at addr, with a preceding count, to a double-number, using the current numeric BASE. If a decimal point is encountered, leave its position in DPL. If numeric conversion is impossible, an error message is given.

OR n1 n2 .... or
Leave the bit-wise logical OR of two numbers.

OUT .... addr
A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display format.

OVER n1 n2 .... n1 n2 n1
Copy the second stack item, placing it as the new top item on the stack.

PAD .... addr
Leave the address of the text output buffer, which is a fixed offset above HERE.

PFA nfa .... pfa
Convert the name field address of an entry, to its parameter field address.

QUERY
Input 80 text characters (or until a "return"), from the keyboard. Text is positioned at the address contained in TIB, with IN set to zero.

QUIT
Clear the return stack, stop compilation, and return control to the operator. No message is given.

R .... n

Copy the top of the return stack to the computation stack.

R# .... addr

A user variable containing the current position of the string editor cursor. See TOP.

R> .... n

Remove the top return stack item, and place it on the top of the computation stack.

REPEAT P,C2

addr n .... (compiling)

Used in a colon-definition in the form:-

BEGIN .... WHILE .... REPEAT

At run-time, REPEAT forces an unconditional branch back to BEGIN. At compile-time, REPEAT compiles BRANCH and the offset from HERE to addr, n is used for error testing.

ROT n1 n2 n3 .... n2 n3 n1

Rotate the top 3 stack items, bringing the 3rd to the top of the stack.

RP!

Initialise the return stack pointer to its implementation start value.

S->D n .... d

Sign extend a single number to make it a double number.

SO .... addr

A user variable containing the initial start value of the computation stack pointer. See SP!

SCR .... addr

A user variable containing the current screen number.

SIGN n d .... d

Stores an ascii "-" sign just before a converted numeric string in the text output buffer when n is negative. n is discarded, but the double number d is retained. Must be used between <£ and £>.

SMUDGE

Used during word definition to toggle the "smudge bit" in the definitions name field. This prevents an un-completed definition from being found in a dictionary search, until compiling is completed without error.

SP!

Initialise the computation stack pointer to its implementation start value.

SP@ .... addr

Return the address of the stack pointer to the top of the stack, as it was before SP@ was executed.

SPACE
Transmit an ascii blank to the selected output device.

SPACES n ....
Transmit n ascii blanks to the output device.

STATE .... addr
A user variable indicating whether compiling or not.

SWAP n1 n2 .... n2 n1
Exchange the two top stack items.

TASK
A no-operation null in the dictionary.

THEN P,C0
addr n .... (compiling)
Used in a colon-definition in the form:-
IF .... ELSE .... THEN
At run-time, THEN serves only as the target of a forward branch from IF or ELSE, and marks the conclusion of a conditional structure. At compile-time, THEN computes the forward offset from addr to HERE and stores it at addr. n is used to trap errors.

TIB .... addr
A user variable containing the address of ther terminal input buffer.

TOGGLE addr 1 b ....
Complement the contents of addr by the bit pattern b

TRAVERSE addr1 n addr2
Move across a variable length name field. addr1 is the address of either the length byte or the last letter. If n=1, the motion is towards high memory, if n=-1, towards low memory. The addr2 resulting is the address of the other end of the name.

TYPE addr count ....
Transmit count characters from addr to the selected output device.

U* u1 u2 .... ud
Leave the unsigned double number product of the two double numbers u1 and u2.

U/ ud u1 .... u2 u3
Leave the unsigned remainder u2 and the unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

UNTIL f .... (run-time)
addr n .... (compile) P,C2
Occurs within a colon-definition in the form:-
BEGIN .... UNTIL
At run-time, UNTIL controls the conditional branch back to the matching

BEGIN. If f is false, execution returns to just after BEGIN, if true execution continues ahead.
At compile-time, UNTIL compiles (0BRANCH) and an offset from HERE to addr. n being used for error tests.

USER n ....
a defining word used in the form:-
n USER cccc
which creates a user variable cccc. The parameter field of cccc contains n as a fixed offset relative to the user pointer register UP for this user variable. When cccc is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of this particular variable.

VARIABLE
A defining word used in the form:-
n VARIABLE cccc
When VARIABLE is executed, it creates the definition cccc with its parameter field initialised to n. When cccc is later executed, the address of its parameter field is left on the stack, so that a fetch or store may access this location.

VOC-LINK .... addr
A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow for control for FORGETing through multiple vocabularies.

VOCABULARY
A defining word used in the form:-
VOCABULARY cccc
to create a vocabulary definition cccc. Subsequent use of cccc will make it the CONTEXT vocabulary, which is searched first by INTERPRET. The sequence "cccc DEFINITIONS" will also make cccc the current vocabulary into which new definitions will be entered. All vocabularies ultimately link to FORTH, and by convention vocabulary names are to be declared IMMEDIATE.

VLIST
List the names of the definitions in the CONTEXT vocabulary. ESCAPE will terminate the listing.

WARNING .... addr
A user variable containing a value used for the control of error messages. If =-1, execute (ABORT) for a user specified procedure, If 0 an error message number is given. See MESSAGE and (ABORT).

WHILE f .... (run-time)
ad1 n1 .... ad1 n2 ad2 n2 P,C2
Occurs in a colon-definition in the form:-
BEGIN .... WHILE(tp) .... REPEAT

At run-time, WHILE selects conditional execution based on the boolean flag f. If f is true (non-zero), WHILE continues execution of the true part through to REPEAT, which then branches back to begin. If f is false (zero) execution skips to just after REPEAT, exiting the structure.
At compile-time, WHILE emplaces (0BRANCH) and leaves ad2 of the reserved offset. The stack values will be resolved by REPEAT.

WIDTH .... addr
A user variable containing the name header maximum length.

WORD c ....
Read the next text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string beginning at the dictionary buffer HERE. Word leaves the character count in the first byte, the characters, and ends with two or more blanks. Leading occurences of c are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the Screen!tape-interface buffers.

X
This is the pseudonym for the "null" or dictionary entry for a name of one character of ascii null. It is the execution procedure to terminate interpretation of a line of text, from either the terminal or screen buffers.

XOR n1 n2 .... xor
Leave the bit-wise exclusive-or of two values.

[
Used in a colon definition to suspend compilation. In the form:-

: xxx [words] more ;

the words after [are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with]. See LITERAL,].

[COMPILE] P
Used in the colon definition in the form:-

: xxx [COMPILE] FORTH ;

[COMPILE] will force compilation of an immediate definition, that would otherwise execute during compilation. The above example will select the FORTH vocabulary when xxx executes, rather than at compile-time.

]
Resume compilation, to the completion of a colon-definition. See [.

GRAPHICS, SOUND, ETC.

This section is intended as a guide to the extra FORTH words provided in this implementation of FORTH for the BBC Micro-computer. It is hoped that sufficient definitions are already provided for the majority of applications, but obviously the user may wish to extend these. The following definitions all refer to DECIMAL numbers, unless otherwise stated.

PLOT - Plot or draw to co-ordinates on the stack. This is identical to the Basic statement, with the exception (as in ALL the following definitions) that the necessary parameters are already on the stack. The format is as follows:-

K X Y PLOT note the use of spaces to delimit the stack values, NOT commas. Users are referred to the BBC manual for a description of operating with various values of K. X and Y are simply the plot co-ordinates.

MOVE - Move to co-ordinates given on the stack. Again similar in action to the Basic statement, but should be used as follows:-

X Y MOVE to move to the graphics origin, for example, we would execute:- 0 0 MOVE

DRAW - Draw to given co-ordinates. Equivalent to Basics' DRAW, and expects two values on the stack:-

X Y DRAW to draw to a point X=600, Y=800 for example, we would execute:- 600 800 DRAW

COL - Define text colour or background. This word expects one value on the stack. For example in a four-colour mode (1 and 5) 1 COL would set the foreground colour to red, and 130 COL would set the background colour to yellow.

MODE - Define screen mode. Used in the format:-

n MODE where n is the screen mode to be selected, e.g. 5 MODE would select mode 5 text and graphics. The amount of memory available for user applications does, of course depend on the mode selected, and in the Model A machine, space is limited to about 1.5K in mode 5, so BE WARNED, selection of a mode which obliterates your program is no joke. Reference should be made to the section describing the memory-map, for details of the system requirements of a particular mode.

GCOL - Define graphics colour. This statement requires two values on the stack, and is identical in effect to the Basic GCOL, use as:-

n1 n2 GCOL n1 can be a number in the range 0-4 and specifies the action as described in the Basic user manual. The value n2 defines the logical colour to be used in future e.g.:-

0 2 GCOL would select colour yellow in mode 5, ready for the next DRAW statement.

DLC - Define logical colour. This is equivalent to the Basic VDU 19 and expects three values on the stack, the third being zero, for anticipated (?) expansion. It should be used as follows:-

n1 n1 0 DLC As an example consider the Basic statement:-
VDU 19,1,4,0,0,0 In FORTH this becomes:-
1 4 0 DLC and the Basic statement:-
VDU 19,0,4;0; becomes:-
0 4 0 DLC in FORTH

DTW - Define text window. This statement is used to re-define the text window on the screen, and expects four values on the stack as follows:-

LeftX BottomY RightX TopY DTW As an example, the Basic statement:-

VDU 28,5,20,30,12 in FORTH becomes:-
5 20 30 12 DTW

DGW - Define graphics window. This is equivalent to the Basic VDU 24, and it expects four values on the stack thus:-

150 300 1100 700 DGW this is directly equivalent to the Basic:-

VDU 24,150;300;1100;700

CLG - Clear graphics area.

CLT - Clear text area.

MTC - Move text cursor. This word is similar in effect to Basics' TAB(X,Y), but the word MTC expects both X and Y already on the stack.

10 12 MTC will move the text cursor roughly to the centre of the screen.

DGO - Define graphics origin. This is equivalent to the Basic VDU 29 and expects the two co-ordinates on the stack:-

X Y DGO So for example to re-define the graphics origin to the centre of the screen:-

640 512 DGO should be executed.

DCHAR - Define a character. This is the FORTH equivalent to the Basic VDU 23 and it expects nine values on the stack, the first being the character number to be re-defined:-

240 28 28 8 127 8 20 34 65 DCHAR will define the little chap used in an example in the Basic users guide.

CHAR - Print out a re-defined character. This is similar to the Basic statement:-

PRINT CHR$(n), and is used in FORTH simply as:-
n CHAR

All of the foregoing definitions make use of the MOS routine OSWRCH, in one form or another, and the following VDU codes are available via the FORTH word EMIT, in each case the equivalent Basic VDU code is given in brackets.

1 EMIT	(VDU 1) send next character to printer only
2 EMIT	(VDU 2) enable printer
3 EMIT	(VDU 3) disable printer
4 EMIT	(VDU 4) write text at text cursor
5 EMIT	(VDU 5) write text at graphics cursor
6 EMIT	(VDU 6) enable VDU drivers
7 EMIT	(VDU 7) make a short bleep
8 EMIT	(VDU 8) backspace cursor
9 EMIT	(VDU 9) forwardspace cursor
10 EMIT	(VDU 10) move cursor down one line
11 EMIT	(VDU 11) move cursor up one line
12 EMIT	(VDU 12) clear text area - equivalent to CLT
13 EMIT	(VDU 13) move cursor to start of current line
14 EMIT	(VDU 14) page mode on
15 EMIT	(VDU 15) page mode off
16 EMIT	(VDU 16) clear graphics area - equivalent to CLG
20 EMIT	(VDU 20) restore default logical colours
26 EMIT	(VDU 26) restore default windows
127 EMIT	(VDU 127) backspace and delete

*FX - Execute Machine Opeating System (MOS) call. A definition of this important call has been made in this implementation, but the user should be warned - a SUBSTANTIAL NUMBER of the *FX functions are not available on early OS Roms. The calls use the routine OSBYTE and are available in FORTH in two distinct ways, first of all via the FORTH word *FX, and secondly as a MONITOR call, (to be described next). The FORTH definition *FX requires three values on the stack - following zeros must NOT be stripped, e.g.:-

n1 n2 n3 *FX Where n1 represents the Accumulator, n2 the X register, and n3 that of the Y register. For example to turn on the cassette motor execute the following:-

137 1 0 *FX and to turn if off:-

137 0 0 *FX A possible motor-control definition could be:-

: MOTOR 137 SWAP 0 *FX ; when typing 1 MOTOR would turn the motor on, and 0 MOTOR turn if off.

The user is referred to the BBC manual for the plethora of functions available via *FX. After making certain OSBYTE calls using *FX, information is carried back to the user via the X and Y registers. Provision has been made to access this information at the following locations:-

Accumulator	- &4CA
X register	- &4CB
Y register	- &4CC

As an example, suppose we wished to ascertain what the lowest address in memory is used by a particular screen mode. A definition could be compiled thus:-

HEX : FMODE (report lowest screen address for given mode)
85 SWAP 0 *FX 4CB @ ; where FMODE would expect the mode number on the stack, e.g.:-
5 FMODE would return on the stack the start address required by screen mode 5.
All other OSBYTE calls can be defined in a similar fashion, referring to the user manual chapter on OSBYTE.

MONITOR - Treat following keyed input as a * MOS command. This word when executed directly at the keyboard should be followed by any of the commands normally preceded by a * in Basic. The *, however should NOT be included. Below are given some examples, and the Basic equivalents are given in brackets for comparison.

MONITOR	FX1	(*FX1) turn on cassette motor
MONITOR	TAPE 3	(*TAPE3) select 300 baud cassette
MONITOR	KEY 0	(*KEY0) programe function key 0.
		For example:-

MONITOR KEY 0 EDITOR L would program the f0 key to make the current vocabulary EDITOR and list the contents of the current FORTH screen buffers. A carriage return could also be included if desired. If it is desired to compile MONITOR into a definition, it should be preceded by the FORTH word [COMPILE] as it is defined as an IMMEDIATE word. On execution from within a colon definition MONITOR will QUERY the operator for a relevant command, and then execute it, before carrying on with the remainder of the definition.

OSWORD - Perform various operating system calls. As with *FX (OSBYTE), the registers of the CPU are used to pass various parameters. The summary of functions with various Accumulator values (n1) is given in the user manual. In FORTH a call to OSWORD is made as follows:-
n1 n2 n3 OSWORD where n1, n2 and n3 are the Accumulator, X and Y registers respectively One demonstration of the use of OSWORD is in the definition of TIME@ which is included in this implementation:-

TIME@ - Read internal elapsed time clock. The definition of TIME@ is as follows:-
HEX : TIME@ 1 EO 4 OSWORD 4EO C@ 4E1 @ 4E3 @ ; where the values 1 EO and 4 are passed in the Accumulator X and Y registers respectively. X and Y are the address at which the call will put the elapsed time reading, and the rest of the definition simply puts these values on the stack for the user.

SOUND, S/TABLE, NOTE and ENV

S/TABLE - Fill a SOUND table with four parameters. This is used as:-
n1 n2 n3 n4 S/TABLE and the parameters are identical to those used in the Basic SOUND statement. Note that S/TABLE merely initialises the values in the table - itself producing no sould.

SOUND - produce a sound, using the parameters in S/TABLE:-
SOUND ....sound (pleasing, hopefullly!)

NOTE - Define and produce a sound. This statement merely combines the two previous ones and is directly equivalent to the Basic SOUND statement, it is used as:-
n1 n2 n3 n4 NOTE   and will produce a sound on execution.

The use of NOTE as well as making a sound, leaves the parameters of the sound in S/TABLE, therefore subsequent replicas of the sound are available via SOUND.
The above definitions being provided separately, allows for maximum speed in the production of a sound, and the user may wish to create similar definitions to cope with the sound channels separately.

ENV - Define a pitch/amplitude envelope for use by SOUND and NOTE. This is identical to the Basic ENVELOPE, and expects 14 values on the stack. Reference should be made to the manual for a description of the individual parameters. The statement:-
2 1 2 -2 2 10 20 10 1 0 0 -1 100 100 ENV   is identical in performance to the Basic:-
ENVELOPE 2,1,2,-2,2,10,20,10,1,0,0,-1,100,100

Other calls may be made via OSWORD, and in each case the 3 parameters should pass the required information as detailed in TIME@. On returning from OSWORD, the values of the Accumulator, X and Y registers are passed to locations &4CD, &4CE and &4CF respectively.

TRIANGLE - Draw a triangle at the co-ordinates given on the stack. TRIANGLE is a high level compound of MOVE and DRAW, and saves considerable typing. It is used as follows:-
X2 Y2 X1 Y1 X0 Y0 TRIANGLE
Executing:- 800 800 800 0 0 0 TRIANGLE   will draw a triangle on the screen at these co-ordinates.

4SIDES - Draw a four-sided figure at the co-ordinates given on the stack, e.g.:-
X3 Y3 X2 Y2 X1 Y1 X0 Y0 4SIDES
Executing:- 0 800 800 800 800 0 0 0 4SIDES   will draw a square, and:-
800 800 0 800 800 0 0 0 4SIDES   a 'twisted' square i.e. 2 triangles with their respective apexes pointing to each other.

3PAINT - Draw and fill a triangle with the stated colour. This word expects 7 values on the stack as follows:-
K X2 Y2 X1 Y1 X0 Y0 3PAINT   where K has a similar effect to that in the PLOT statement.
Executing:- 5 MODE 0 1 GCOL 85 600 600 1200 0 0 0 3PAINT will draw and fill a red triangle.

4PAINT - Draw and fill a 4 sided figure. This word expects 9 values on the stack as follows:-

K X3 Y3 X2 Y2 X1 Y1 X0 Y0 4 POINT where K has similar effect to that in PLOT and 3PAINT.

Executing:- 5 MODE 0 2 GCOL 85 0 800 800 800 800 0 0 0 4PAINT will draw and fill a square in logical colour 2 at the co-ordinates given.

RANDOM NUMBERS
(RND), RND, ABSRND and SEED

(RND) - Generate a random number in the range +32767 and -32768. This is a machine-code primitive 33 Bit Psuedo-Shift-Register-Generator, or PSRG for short. If (RND) is executed, it will leave a random number in the range given above, on the stack. This of course is too large a range for most purposes, and so the next definition is provided to limit the range of numbers produced.

RND - Generate a random number with a maximum value as stated on the stack. Used as:-

n RND where RND will produce a number up to a maximum n, although negative numbers will also be produced.

ABSRND - Generate a random number with a maximum value as stated on the stack, the number will be positive and in the range 0 - n. Use as:-

n ABSRND

SEED - Initialise the primitive (RND). At switch-on the primitive (RND) is initialised to zero, and if a different sequence of random numbers is required SEED may be used to restart the primitive thus:-

n SEED where n can be any number in the FORTH integer range.

The mention of sequence when talking of random numbers may seem a little odd, in fact there is sequency in the primitive described, it repeats itself after some 8 million numbers have been generated, and for most purposes this can be regarded as random!

A further possible extension to the random number definitions could be that of R/RND where R/RND would produce random numbers between a minimum and maximum value as stated on the stack. The definition of R/RND is not included in this implementation - yes we have left something for you to define! So here is one possible definition of R/RND:-

```
: R/RND ABSRND 2DUP > IF + ELSE SWAP DROP THEN ;
```

This definition can be used as follows:-

lower-limit upper-limit R/RND Three examples follow:-

```
12 24 R/RND . 13 OK
 3 24 R/RND .  5 OK
12 24 R/RND . 18 OK
```

SYSTEM DESCRIPTION

It is important to the user of FORTH, that He/She is aware of the allocation of memory for various system operations. Below are all important addresses and their use. Note that the addresses are all in HEX.

Zero page.
0-7 not used
8-A random number shift register
8-70 FORTH stack

78-7F	N	address of 8 byte scratchpad
80-82	IP	interpretive pointer
83-84	W	code-field pointer
85-86	UP	pointer to user-area
87	X	temp storage for X register of CPU (X-Save)
8E-8F		cold/warm flags (not used by FORTH)
90-FF		operating system use only

Page 1 and above.
100- upwards FORTH Terminal input buffers
1FF- downwards CPU return stack

200-3FF	operating system use only
400-47F	FORTH user-variables
480-491	booted entry for FORTH
492-4A1	booted entry for (ABORT)
4A2-4B4	booted entry for EDITOR
4B5-4C9	booted entry for GRAPHICS
4CA-4CC	returned parameters from *FX (OSBYTE) ,
4CD-4CF	returned parameters from OSWORD
4D8-4FF	Free - not used by FORTH
500-505	Intersect for 'MESSAGE'
506-6FF	Free - not used by FORTH
700-77F	command-line buffer for MONITOR, SAVE, LOAD and LIST
780-7FF	reserved for floating point extension
800-DFF	operating system use only
E00-121F	tape/screen editor buffers

1220- upwards Start of user-application area. (HERE on COLD start)

The following are the important machine-code calls in the FORTH system. These will be required by the user when compiling primitives using CREATE.

8240 - PUSH Push the Accumulator as high byte, CPU return stack as low, onto the FORTH stack
8242 - PUT replace stack items with above
8247 - NEXT execute the next FORTH address incrementing IP
8272 - SETUP move n items from stack to scratchpad (N)
82CE - BUMP move IP by 2 if test is true
835A - POPTWO drop 2 stack items
835C - POP drop 1 stack item
8545 - PUSH0A push zero as high byte, Accumulator as low, onto the FORTH stack

Note that the only sub-routine is that of SETUP, all other calls should be jumps.

THE FORTH USER-VARIABLES

Below are the addresses of the user-variables. Users wishing to define nw user-variables should make the relevant offset into the area not less than &32, so that conflicts do not occur.

Address (in HEX)	Name	Description	Contents at COLD(HEX)
400	NTOP	top word in FORTH	95E8
402	RUBOUT	backspace and delete	7F
404	UAREA	start addr. of this area	400
406	TOS	top of FORTH stack	70
408	TOR	top of return stack	1FF
40A	TIB	terminal input buffer	100
40C	WIDTH	max name header	1F
40E	WARNING	control warning modes	0
410	FENCE	lower limit for FORGET	1220
412	DP	dictionary pointer	1220
414	VOC-LINK	new vocabulary pointer	490
416	BLK	interpretation flag	-
418	IN	offset into source text	-
41A	OUT	display cursor position	-
41C	SCR	current screen number	-
41E	OFFSET	for possible disc use	-
420	CONTEXT	vocabulary 1st searched	48E
422	CURRENT	compile into, 2nd searched	48E
424	STATE	compile/execute flag	0
426	BASE	current number i/o	0A
428	DPL	decimal point location	-
42A	FLD	output field width	-
42C	CSP	check stack position	-
42E	R#	screen editor cursor	-
430	HLD	addr. of last char in PAD	-

The first free offset into the user-area is &32, and the maximum allowable offset is &7E. There is therefore, space for up to 39 extra user-variables - if required!

Square, Diamonds etc.

Square

The implementation supports the definition 4SIDES, which can, of course, be used to draw a square. As an alternative the following definition may be compiled. SQUARE expects the size of the side on the stack.

```
: SQUARE Ø Ø MOVE    (Move to graphics origin)
  DUP Ø DRAW         (Draw bottom horizontal)
  DUP DUP DRAW       (Draw R H vertical)
  Ø SWAP DRAW        (Draw top horizontal)
  Ø Ø DRAW           (Draw L H vertical)
;
```

Now executing:- 5 MODE 640 512 DGO Ø 2 GCOL
200 SQUARE

Will draw a square starting and finishing at the graphics origin in the middle of the screen. A negative number will draw the square towards the LH bottom portion of the screen.

To draw a number of square of differing sizes but with the same origin, the following may be tried:-

```
: SQS Ø DO DUP I * SQUARE LOOP DROP ;
```

SQS expects the following on the stack:- incr no
Where incr is the increment for the next square drawn and no is the number of squares to draw.

The above examples both draw squares with one corner sited at the graphics origin. An alternative to this is the method of drawing the square with the graphics origin at its centre. For this, half the co-ordinates need to be negative and the length of side is effectively doubled.
Consider the following steps:-

a) -X -Y MOVE
b) X -Y DRAW
c) X Y DRAW
d) -X Y DRAW
e) -X -Y DRAW

Since X and Y have the same magnitude, i.e. ½side, a definition can be compiled as follows:-

```
: SQ DUP MINUS DUP 2DUP    (Duplicate twice for MOVE and
                            last DRAW statement)
  MOVE                     (Move to -X,-Y)
  ROT DUP DUP MINUS        (get side and duplicate twice,
                            make Y negative)
  DRAW                     (Draw X,-Y)
  DUP DUP DRAW             (Draw X,Y)
```

```
            OVER SWAP DRAW              (Draw -X,Y)
            DRAW                        (Draw -X,-Y)
          ;
```

Executing:- 640 512 DGO 200 SQ will draw a square of side 400 in the exact centre of the screen.

The definition 4SIDES can be used as an alternative to the above as follows:-

```
          : SQ2
            DUP MINUS SWAP DUP 2DUP DUP MINUS
            DUP DUP 4SIDES ;
```

it also expects the side of the square (½ of actual size) e.g.:-

```
          n SQ2
```

and will have identical action to that of SQ, it will however, be a little slower in execution, than the previous example of SQ.

Diamond

With some rearrangement, similar methods can be derived to draw a diamond on the screen. The necessary steps in this case are as follows:-

```
a)  -X   0 DRAW
b)   0   Y DRAW
c)   X   0 DRAW
d)   0  -Y DRAW
e)  -X   0 DRAW
```

and a suitable definition could be:-

```
          : DIA
            DUP MINUS Ø MOVE
            Ø OVER DRAW
            DUP Ø DRAW
            Ø OVER MINUS DRAW
            MINUS Ø DRAW
          ;
```

Now executing:- 5 MODE 640 512 DGO 200 DIA will draw the diamond in the centre of the screen.

'Persian' type patterns can be created with the combination of squares and diamonds. One such possibility is:-

```
          : PERSIAN (step/mix.......)
            2DUP Ø DO 4 ABSRND 8 ABSRND GCOL
                      I SQ DUP +LOOP DROP
                  Ø DO 4 ABSRND 8 ABSRND GCOL
                      I DIA DUP +LOOP DROP
          ;
```

Executing:- 2 MODE 640 512 DGO 8 500 PERSIAN will draw the carpet measuring 1000 x 1000 on the screen. The range of carpets available can be demonstrated in the definition:-

```
          : CARPETS BEGIN 8 500 PERSIAN ?ESC UNTIL ;
```

and if CARPETS is executed e.g.:-

```
2 MODE 5 EMIT 640 512 DGO CARPETS
```

will demonstrate the carpets until the ESCAPE key is pressed.

The user can experiment with various step sizes (used by +LOOP), which will affect the density of weave, the values in the example are a compromise between definition and speed.

Circles

Since FORTH does not normally support floating-point arithmetic, an alternative to the use of SIN and COS for circle-drawing must be used.

One such method is that of iteration, which simply draws a series of short lines until the line bumps into itself. Before this can achieved, a definition must be entered to query the graphics screen. This is directly equivalent to the BASIC POINT, and requires the setting up of a parameter block thus:-

```
0 VARIABLE PBLOCK 3 ALLOT
```

Now to the definition of POINT which expects the X and Y co-ordinates on the stack.

```
: POINT   (X/Y.........)
  (Query screen at given co-ordinates leaving the logical colour
   found there on the stack)
  PBLOCK SWAP OVER 2+ !   (Store Y)
  SWAP OVER !             (Store X)
  9 OVER 256 /MOD OSWORD
  (get address of PBLOCK in X and Y Registers of CPU and query
   point on Screen)
  4 + C@   (put value on stack)
;
```

Now the definition of circle:-

```
: CIRCLE   (X/Y/S.........)
  >R 2DUP MOVE
  BEGIN 2DUP DRAW SWAP OVER R / + SWAP OVER
  R / - 2DUP POINT Ø= WHILE REPEAT
  R> DROP 2DROP ;
```

Now executing:-

```
1 MODE 640 512 DGO 300 200 20 CIRCLE
```

will draw a circle in the centre of the screen, where X,Y is the starting point and S determines the size of steps round the circle.

Get Scrolled!

Readers of the BBC Users Manual may have noticed that the VDU 23 calls have a dual purpose. They may be used to define characters, or to converse with the 6845 CRTC. In FORTH, the equivalent procedure is DCHAR and expects 9 parameters on the stack.

When using DCHAR to program the CRTC, the following format should be adopted.

```
ØRXØØØØØØDHCAR
```

where R is the register in the CRTC you wish to address, and X is the value to be loaded into it.

To scroll the screen sideways for instance, the following definitions can be entered:-

```
: SCROLL Ø 13 ROT ØØØØØØ DCHAR ;
```

Where SCROLL expects a parameter on the stack. To scroll the screen left 10 places for instance, execute:-

```
10 SCROLL
```

and to move it right 5 places from this position, execute:-

```
5 SCROLL
```

Recursion

Recursive routines can be extremely useful, achieving a large work outout for quite short programs.
A recursive definition is one that contains references to itself, and obviously whilst compiling such definitions, the entry cannot find a reference to itself, so a trick is used to enable this.
An IMMEDIATE word MYSELF, is defined as follows:-

```
: MYSELF LATEST PFA CFA , ; IMMEDIATE
```

and an example of the use of MYSELF, is that of a recursive definition in the calculation of Factorials.

```
: (FACTORIAL)
  -DUP IF DUP ROT * SWAP 1 - MYSELF THEN ;
: FACTORIAL (n.......)
  DUP 0< OVER 7 > OR 5 ?ERROR
  1 SWAP (FACTORIAL) . ;
```

In the above, calculation of the factorial is done by (FACTORIAL) whilst range checking is carried out by FACTORIAL, which then calls (FACTORIAL) to do the calculation.
The highest number which can be processed is 7, over this arithmetic over-flows would occur. The reader may wish to experiment with double-number arithmetic to increase the range of numbers handled.

Executing:- 3 FACTORIAL gives <u>6 OK</u>
4 FACTORIAL gives <u>24 OK</u>
5 FACTORIAL gives <u>120 OK</u>
6 FACTORIAL gives <u>720 OK</u>
7 FACTORIAL gives <u>5040 OK</u>

Case

In the following example of a positional CASE, execution of a particular word can be selected by the value on the stack. The value must be in the range 0 - (n-1) where n is the number of cases included in the definition.

```
: CASE
        <BUILDS SMUDGE ]
                DOES> SWAP 2* + @ EXECUTE ;
```

One possible use for CASE statements is the extension of the error message handler in FORTH.

An intersect is provided at &502, which at present, simply prints out the stack value corresponding to the error type.

The extension error handler could be formatted as follows. Each error message is given its own definition e.g.

```
: ØERROR ." Unrecognised word" ;
```

and these are included in a CASE definition as follows:-

```
CASE ERRORS ØERROR 1ERROR 2ERROR ....... ;
```

Now a new message handler is defined:-

```
: NMSG DUP . (Duplicate and print error number)
  ERRORS ;   (Report from error case)
```

The intersect can be pointed to the new message handler with

```
HEX        ' NMSG CFA 502 !
```

and when an error is detected a full error msg will be printed out

e.g. . ?MSG # 1 stack empty

Arrays, tables

One definition for a one-dimensional array is:-

```
: ARRAY <BUILDS 2* ALLOT
        DOES> SWAP 2*+ ;
```

A multi-element array of any size can now be defined e.g.:-

```
20 ARRAY PARAMETERS
```

The array called PARAMETERS may be be filled thus:-

```
 9 Ø PARAMETERS !
21 1 PARAMETERS !
```

This will put the values 9 and 21 into the first 2 elements of the array, and the contents of a particular element can be accessed with:-

n PARAMETERS @   putting the value on the stack, where n is the element number, or:-

n PARAMETERS ?   which will print the value out.

A two-dimensional Array can be set up if the following is defined:-

```
: 2ARRAY <BUILDS DUP , * 2* ALLOT
  DOES> ROT OVER @ * ROT + 2*
  + 2+ ;
```

and the array defined e.g.

```
14 6 2ARRAY BOX
```

to create a 14 x 6 array called BOX.

The contents of a particular element can be accessed in a similar fashion to that of ARRAY, except that 2 values are required on the stack e.g.:-

n_1 n_2 BOX @

For instance:- 2 9 BOX @ will push the contents of the 2,9 element onto the stack.

Further OSWORD calls

Because of the dictates of space, OSWORD has only been used in the definition of SOUND, ENV & TIME. Further calls may of course be entered by the user, and we have seen the compilation of one such call in that of POINT.

Since OSWORD expects a parameter block for operation, this can be set up for each discrete definition as a variable and extra bytes by the use of VARIABLE & ALLOT.

As an example consider the OSWORD call with Accumulator = 10, which will read a characters' definition and requires a 9 byte parameter block as follows:-

```
XY        character required
XY+1      top row of displayed character
              .
              .
              .
              .
              .
XY+8      bottom row of displayed character
```

The block may be set up as follows:- Ø VARIABLE CBLOCK 7 ALLOT
and the definition:-

```
: RCD   (Read Character Definition)
 CBLOCK  SWAP  OVER  C!   (Put characters in 1st location of
                           CBLOCK)
 10 SWAP 256 /MOD OSWORD ;
 (get address in X & Y registers of CPU and do OSWORD call)
```

This can be now used as:-

```
c RCD    where c is the ASCII value of the character
```

e.g. executing:-

```
65 RCD
```

will resolve the character definition of 'A'
To read out the character parameters the following can be entered:-

```
: RCBLOCK   (Read CBLOCK contents)
 9 Ø DO CBLOCK I + C@ . LOOP ;
```

Now executing RCBLOCK will give:-
65 6Ø 102 102 126 102 102 102 Ø

You may wish to check the validity of this procedure by executing:-
5 MODE 225 60 102 102 126 102 102 102 Ø DCHAR
and then try:- 225 CHAR
when a capital 'A' should result.
The parameters of a character may be trebled so that they occupy 3 character positions for instance:-
225 60 60 60 102 102 102 102 102 DCHAR
226 102 126 126 126 102 102 102 102 DCHAR

227 102 102 102 102 102 Ø Ø Ø DCHAR
and execute:- CR 225 CHAR CR 226 CHAR CR 227 CHAR
this will print a treble-size 'A' on the screen.
The text cursor could of course be moved with the MTC command.

POS & VPOS

Similar statements to the Basic POS and VPOS, to give the horizontal and vertical positions respectively of the text cursor, can be defined with the use of *FX. Note that the returned parameters (when applicable) from this call are situated in &4CA-4CB.
The definition:-

```
: CTC   (Current Text Cursor)
  134 Ø Ø *FX ;
```

will on executing CTC push the X & Y co-ordinates of the text cursor into &4CB and &4CC respectively. These can then be accessed together or independantly with suitable definitions.

```
HEX        : POS CTC 4CB C@ ;
           : VPOS CTC 4CC C@ ;
```

POS & VPOS will leave the respective horizontal and vertical positions on the stack, and a further definition:-

```
HEX        : CCP   (Current Cursor Position)
             CTC 4CB C@ 4CC C@ ;
```

will push both values on the stack.
The examples that follow demonstrate how to print the contents of a string variable vertically on the screen.
First we define the following:-

```
: STRING <BUILDS DUP C, Ø C, ALLOT
  DOES> 1+ ;

: $IN HERE C/L 1+ BLANKS
  1 WORD HERE PAD C/L 1+ CMOVE
  PAD DUP C@ 1+ ;

: $! 2DUP 1- C@ 1+ >
  IF
  CR ." $ Too large" 2DROP DROP QUIT
  THEN
  SWAP CMOVE
;

: $@ COUNT ;
```

Now execute:-

```
8 STRING LETTERS
$IN VERTICAL
LETTERS $!
```

This puts "VERTICAL" into the string variable LETTERS

Now a definition to print our the string vertically:-

```
: VTYPE $@ 1+ Ø DO DUP I + C@
  CCP SWAP 1 - SWAP 1+ MTC EMIT LOOP
  DROP ;
```

Executing LETTERS VTYPE will print out the following:-

```
V
E
R
T
I
C
A
L
```

VTYPE can be defined so to include it in a definition to print vertically in direct or compiled mode, without the use of string variables e.g.

```
: VTYPE -DUP IF OVER + SWAP
  DO I C@ CCP SWAP 1 - SWAP 1 + MTC
  EMIT LOOP ELSE DROP THEN ;

: (V") R COUNT DUP 1+ R> + >R VTYPE ;

: V" 34 STATE @
  IF
    COMPILE (V") WORD HERE C@ 1+ ALLOT
  ELSE
    WORD HERE COUNT VTYPE
  THEN
;   IMMEDIATE
```

After compiling the above try:-

```
CLT V" VERTICAL STRING "
```

Immediately the return key is pressed the string is printed out vertically.
Now try:- `: V$ V" VERTICAL PRINTING " ;`
and execute:- `CLT V$`
when again the string will be printed vertically.
The above three definitions demonstrate the tremendous advantage of FORTH in compiling new print formats.
Note that no check is given on the length of the string, and if this overruns the bottom of the screen, the rest of the text will be printed out horizontally. The user could, of course, test the current Y cursor position inside the definition of VTYPE and scroll the screen if necessary.

Cubes

3-dimensional cube-drawing can be achieved without the use of Polar-co-ordinate plotting, after all a cube can be represented by 2 offset squares with the respective corners joined by lines. The front face of the cube may be described by the co-ordinates f0, f1, f2 and f3 while the back face

equivalents as:-
b0, b1, b2 and b3. The vector displacement to give a 3D effect may be represented by VX and VY.

Possibly the easiest method of drawing a cube is to store the necessary co-ordinates in an array as these are calculated, rather than attempt complicated stack manoeuvres.

Since each point visited on the front face requires re-visitation we can make use of ARRAY as described earlier to store these, it is also useful to define the vector displacement Vx and Vy as variables and to make the starting Graphics origin a pair of variables also.

The following definition assumes that the user has entered the definition of SQ also, as this is used to draw the back face of the cube.

```
Ø VARIABLE VX
Ø VARIABLE VY
8 ARRAY  FACE
2 ARRAY  ORIGIN

: VPLOT  (relative line from fn to bn)
  1 VX @ VY @ PLOT ;
```

Now to the definition of cube itself, which expects the size of side (to be doubled) on the stack.

```
: CUBE                          (side/..........)
  Ø ORIGIN @ 1 ORIGIN @ DGO
  DUP MINUS DUP 2DUP 2DUP
  1 FACE ! Ø FACE !             (save point f0)
  MOVE
  ROT DUP DUP MINUS 2DUP
  3 FACE ! 2 FACE !             (save point f1)
  DRAW
  DUP DUP 2DUP
  5 FACE ! 4 FACE !             (save point f2)
  DRAW
  OVER SWAP 2DUP
  7 FACE ! 6 FACE !
  DUP >R                        (save side on return stack)
  DRAW DRAW
  R>                            (bring back side)
  Ø ORIGIN @ VX @ +             (calculate new X origin)
  1 ORIGIN @ VY @ +             (calculate new Y origin)
  DGO SQ                        (and definition then as new
  Ø ORIGIN @ 1 ORIGIN @ DGO      graphics origin-do a square)
  Ø FACE @ 1 FACE               (get f0)
  MOVE VPLOT                    (move there and then plot to b0)
```

```
        2 FACE @ 3 FACE @
        MOVE VPLOT                  (ditto for f1)
        4 FACE @ 5 FACE @           (ditto for f2)
        MOVE VPLOT
        6 FACE @ 7 FACE @
        MOVE VPLOT                  (do for last point f3)
      ;
```

Now executing the following:-

```
      100 VX !
      100 VY !
      640 0 ORIGIN !
      512 1 ORIGIN !
      5 MODE 200 CUBE
```

When you should be rewarded with a cube with the front face in the centre of the screen.

The user can experiment with various values of VX and VY both positive and negative, and below is given a definition for drawing 4 cubes with vector displacements of both signs:-

```
      : 4CUBE DUP CUBE VY @ -1 +- VY !
              DUP CUBE VX @ -1 +- VX !
              DUP CUBE VY @ -1 +- VY !
              DUP CUBE VX @ -1 +- VX !
      ;
```

Now executing:- 200 4CUBE will draw the four cubes on the screen. The principle can be extended to pattern drawing, if desired with:-

```
      : PCUBE Ø DO 4 ABSRND 15 ABSRND GCOL
        I 4CUBE DROP DUP +LOOP DROP
      ;
```

and if this is executed e.g.

1234 SEED 8 500 PCUBE will build up a multi-coloured pattern on the screen.

New Fill routines in 1.0 operating system.

The 1.0 operating system contains a set of primitives to fill almost any desired shape with colour using the following PLOT statements. The action is to search along an X axis left and right whilst the pixels found are in the current background colour. The two points X1,Y and X2,Y then have a line drawn between them as per the relevant PLOT code K, where K may be any of the following:-

72	line is not drawn	Relative
73	line is drawn in foreground colour	Relative
74	line is drawn in inverted	Relative
75	line is drawn in background colour	Relative
76	line is not drawn	Absolute
77	line is drawn in foreground colour	Absolute
78	line is drawn inverted	Absolute

79 line is drawn in background colour Absolute

Of these new PLOT commands, possibly 77 and 78 are the most useful.

To fill a shape with colour, we must specify the start value of the X co-ordinate, generally this will lie in the centre of the shape to fill, and specify the lower and upper limits of Y.

Two definitions follows. One fills a desired shape with colour, the other inverts the colour already there.

```
: PAINT      (X/Yupper/Ylower........)
  DO 77 OVER I PLOT 4 +LOOP DROP ;
: INVERT     (X/Yupper/Ylower........)
  DO 78 OVER I PLOT 4 +LOOP DROP ;
```

Consider the following irregular polygon.

```
2 MODE 640 512 DGO
-200 -100 MOVE
-300 -200 DRAW
500 -200 DRAW
500 Ø DRAW
-300 Ø DRAW
-200 -100 DRAW
```

This draws a ribbon on the screen, and this may be coloured with:-

```
Ø 1 GCOL Ø Ø -200 PAINT
```

Reading characters on the screen

It may be required (for a bubble sort for instance) to read the ascii value of a character at a particular position X,Y on the screen. If the user has previously defined CCP, as discussed earlier we can do this quite easily. First of all the definition to query the screen at the given cursor position.

```
HEX          : ?CHAR 87 Ø Ø *FX 4CB C@ ;
```

now the definition to save the present cursor position, move to the given position, querying the character and then returning to the present cursor position to print out the result:-

```
DECIMAL      : ?XY
               CCP >R >R  (put on return stack the current cursor position)
               MTC        (move to position given)
               ?CHAR      (get ascii value of character there)
               R> R> MTC  (move back to current position)
                          (and print out value of character found)
               ;
```

The above is used simply as:-

```
X Y ?XY
```

Suppose the following is typed when the character at 0,5 = "A":-

```
0 5 ?XY
```

The result will be 65 OK printed after ?XY

Of course the . in ?XY can be left out if it is required to leave the ascii value found on the stack.

Un-Compilation

It may be useful on occasions to be able to un-compile COLON definitions i.e. re-assemble them in source form, and below is one such un-compiler. Note that it cannot handle ." (dot-quote). This has been left to the user to try.

```
HEX
            : WASIT? = SWAP DUP ;
            : DELTA DUP             (duplicate address)
              @                     (get code address)
              2+                    (change to parameter field address)
              DUP                   (duplicate for test)
              826C =                (check for CLITERAL)
              IF                    (Cliteral)
              DROP 2+ DUP C@ . 1+   (print out the literal value)
              ELSE DUP
                      ' LIT WASIT?
                      ' BRANCH WASIT?
                      ' ØBRANCH WASIT?
                      ' (LOOP) WASIT?
                      ' (+LOOP) WASIT?
              DROP DROP + + + +     (add up flags)
                      IF 2+ DUP @ U. CR
                      THEN
              THEN
            ;
            : UN: -FIND IF (found) DROP (header count)
                      DUP (code address)
                      2 - @ 873F = (Docolon)
              IF
                BEGIN DUP @ 2+ ' ;S = 0=   (is it not ;S ?)
                WHILE DUP @ 2+ NFA ID. DELTA 2+
                REPEAT
                ." ;S "
              ELSE ." Not a colon  defition " QUIT
              THEN
              ELSE (it not found) 2DROP ." Can't find it ! " QUIT
              THEN
            ;
```

The un-compiler should be used as:-

UN: name

An example of machine output for a typical un-compilation follows:-

```
HEX          UN: DELTA DUP @ 2+ DUP LIT 826C
             = ØBRANCH 12
             DROP 2+ DUP C@ . 1+ BRANCH 3C
             DUP LIT 822F
             WASIT? LIT 82A7
             WASIT? LIT 82C6
             WASIT? LIT 82E7
             WASIT? LIT 8317
             WASIT? 2DROP + + + + ØBRANCH C
             2+ DUP @ U. CR ;S OK
```

A Forth Assembler

Users of FORTH will be familiar with the compilation of machine-code within the definition of a word. Generally, this is achieved by the use of the compiling words , and C, which respectively remove two or one bytes from the stack and compile these into the dictionary.

The use of , stores the two bytes in reverse order and the resulting code, before compilation looks very strange indeed. The method does however have the advantage of speed and less typing than that of the use of mnemonics, it does of course pre-suppose a good working knowledge of the necessary op-codes and this is where a good mnemonic assembler excels. For instnce the 6502 instruction CMP has eight possible op-codes, each for a particular addressing mode. It is therefore far easier to type CMP and let the assembler sort out which actual op-code is required.

Assemblers for FORTH usually follow the FORTH tradition of pushing operands onto the stack and then processing these with a mnemonic instruction so that the correct bytes are installed in memory. Again, the resulting text is awkward to read and instal and a better method, closely related to traditional assembler format was sought by the author. The following FORTH definitions are the result of recent work in this direction, and although they relate to the 6502 CPU, it would be a relatively simple matter to change them to suit other processors. Fanatical FORTH addicts could probably improve on many of the definitions, I have kept these as easy to follow as possible, as many would-be users may be new to FORTH. The definitions can be compiled directly, but better still use made of the screen editor, both to compile them and save them on disc/tape.

Compiling the Assembler

The definitions have been given here in the form of screens, and when entered, the bracketed comments may be left out if desired.

SCR101, PUSH, PUT, NEXT etc.

It is convenient to give the important machine-code calls to FORTH a name, this has been done for PUSH, PUT etc. Also since these are all jumps, the instruction 4C will be compiled into the dictionary together with the address.

Instructions

The instruction set of the 6502 can be divided into 3 main groups as follows:-
Group a) Single-byte instructions e.g. TAX, CLC etc.
Group b) Branch instructions e.g. BNE, BVS etc.
Group c) Memory operatives e.g. LDA, STX, EOR etc.
Since group a) does not require an operand, the definition can act to compile the relevant op-code directly into memory. This is provided for by INS. In addition to the single byte instructions the op-codes for JSR, JMP and JMI (JMP indirect) having only one valid addressing mode, may be defined in the same manner. The branch op-codes are also of one value only and these are similary defined.

SCR102

The artificial program counter 'P' is defined, as this can save some typing. It simply puts the value of HERE on the stack. The next four definitions are concerned with forward and backward branches, and a simple check is made to ensure that these are not too large. Note also that the definitions >> and << both calculate the branch from the branch-instruction itself, as in normal assembler practise.

We now move to group c), and a convenient method for their compilation, is to define each instruction as a table of 9 values. The format given in the screen is as follows:-

,X)),Y A,Y Z,Y A,X ABS ZP IMM

where the columns are possible addressing modes, and the rows (1 row per line of screen) refer to a particular Mnemonic.
Valid codes are inserted at the relevant intersect, and a zero where an addressing mode is not supported.
The definition INSTR builds the table body for each instruction and when executed puts the start address of the table on the stack. The 9 values are then inserted into the instruction body with:-

(INSTR) 9 CFILL

The remainder of SCR 104 is concerned with error trapping and address size etc.

SCR 105 Compiler Directions

These definitions are concerned with examining the value of the operand and calculating the offset into the instruction body to access the correct op-code for a particular addressing mode. The FORTH "," is also re-defined as F, to avoid confusion during assembly, and the definition CODE is included, as the user may wish to compile a headerless primitive.
Having got this far we can now use the assembler and below is an example of use within a CREATE definition:

```
ASSEMBLER  HEX  CREATE  2/  (fast divide by 2)
  CLC  LDA  1  ,X  BPL  ØB  >>
  INC  0  ,X  BNE  4  >>  INC  1  ,X
```

```
BEQ 3 >> SEC ROR 1 ,X
ROR 0 ,X NEXT SMUDGE
```

When entered as above the compiled code will be:-

```
18
B5 Ø1
1Ø Ø9
F6 ØØ
DØ Ø2
F6 Ø1
FØ Ø1
38
76 Ø1
76 ØØ
4C 47 82
```

The action of 2/ is identical to that of:- 2 / but is about sixty times as fast !

Macros

A macro may be defined within a colon-definition e.g.

```
: 4LSRA   (logically shift right 4 times)
  LSRA LSRA LSRA LSRA ;
```

So that when the macro is executed it compiles the relevant machine-code into memory. Such macros can save considerable typing, and the way is also clear to conditional assembly using IF....ELSE....THEN, and similar constructs. One further commonly used combination is:-

```
: LMASK   (mask lower 4 bits of accumulator)
  AND FØ # ;
```

and further possibilities are the extension of the instruction set with pseudo-instructions, such extensions are limited only by the users' imagination, and the dictates of memory space.

Errors

Hopefully you will not type any, but if you do, two traps are included as previously mentioned. Some users may wish to further improve on this, although the author has found those provided quite adequate. The following are brief examples of the traps in operation.

STA 80 # will give:-
Illegal instruction preceding 80 it is always illegal to attempt to store in immediate mode.
LDA 8000),Y will give:-
Operand 8000 too large the post-indexed indirect instruction is not available for absolute addresses.
STX 56 ,X gives:-
Illegal instruction preceding 56 obviously !!! but it can occur through mistyping.

The above traps may not necessarily have conveyed exactly the correct message in all cases. For instance the directive:- # in STA 80 could itself be correct, the error being in the instruction STA, where maybe LDA was intended. They will, however flag a bad instruction combination, and give sufficient information for you to discover why your assembler stopped assembling.

To summarise the directives and their use, given below are the legal possibilities in each instruction group.

Group a) Single op-code instructions NO DIRECTIVE REQUIRED

Group b) Branch instructions
Valid directives for the group are << and >> preceded by a value on the stack.

Group c) Multiple op-code instructions
These use the directives , ,X ,Y ,X) and),Y all of which are preceded by a value on the stack. The values required can of course be put there in any suitable way, for instance a constant typed in, or a variable coupled with the word @
The instructions JMP, JMI and JSR will be followed with a value and then the FORTH compiler directive F,

Strings

Strings of ascii code may be compiled into the machine-code if the following definition is entered:-

```
: ASCII   (compile ascii string in-line with code)
  1 WORD HERE DUP 1+ SWAP DUP C@
  DUP ALLOT CMOVE  ;
```

It should be used e.g.:-

ASCII (string) when the word or words after ASCII will be compiled into the dictionary.

Finally you may wish to read the machine-code produced by your assembly, here is one simple possibility.

```
ASSEMBLER HEX   : READ FF 0 DO CR DUP I + DUP
                  U. C@ . KEY DROP LOOP DROP ;
```

Used as:- addr READ

READ will read out the address followed by its contents, pressing a key will step the read-out, and you can end the reading by pressing the ESC key.

SCR 106

ACTIVE LABELS

Methods have been employed in most Basic assemblers, to enable the user to specify a branch or jump, to a label - even of the label address is not known. Such assemblers normally require two passes for successful compilation of the code.

Such schemes may also apply to the FORTH assembler, and the definitions on SCR 106 are one such possibility. It must be stressed that the labels so produced can only be referred to once in the assembly, and further work needs to be done in this direction.
An 'active' label is defined consisting of a name header and 4 byte parameter body, together with a series of conditional actions. The parameter body is divided into 2 two-byte areas, which I shall call PF1 and PF2. The label is executed in two distinctly different ways.

a) it is ESTABLISHED
b) it is USED

Considering case a) first, the label is initialised with the address of HERE when it is executed with a stack value of 1. It then tests to see if it has been used, and if so, does the necessary calculation and compilation of addresses. Now turning to case b) the label checks to see if it has been established or not. If the latter is true the label examines the instructions and allots space accordingly, the address of this space being stored in PF2, for further reference. If the label has been established, it performs the necessary calculation and again compiles the correct address form into the dictionary. Case b) is selected by a zero on the stack prior to executing the label.

To summarise:-

1 label (Establish label HERE)
Ø label (Use label HERE)

If the contents of SCR 106 have been entered we can create labels as follows:-

```
LABEL name
LABEL different-name
```

and as many labels as are deemed necessary may be defined in this way. Each one after creation contains zero in PF1 and PF2, ready for their use in assembly e.g.

```
HEX FORTH DEFINITIONS ASSEMBLER
LABEL CRCLP
LABEL NOC
CREATE (CRC)     (Cyclic redundancy check)
LDA 0 ,X
STX 87 ,
LDX 08 #
1 CRCLP LSRA
ROL 0 ,
ROL 1 ,
BCC 0 NOC
PHA
LDA 0,
EOR 2D #
STA 0 ,
PLA
1 NOC DEX
```

```
                BNE 0 CRCLP
                LDX 87 ,
                POP SMUDGE
```

After the above compilation, the following machine-code will result:-

```
B5 00
86 87
A2 08
4A
26 00
26 01
90 08
48
A5 00
49 2D
85 00
68
CA
D0 EE
A6 87
4C 5C 83
```

The primitive can be initialised and used to find the CRC of any sized ROM, e.g. for 4K ROMS :-

```
HEX           : CRC 0 0 ! 1000 0 DO DUP I + C@
                (CRC) LOOP DROP 0 @ U. ;
```

and executing:- HEX 8000 CRC

and:- 9000 CRC

will give 77C0 and B4E5 for the lower and upper portions of the V2.0 ROM

```
EDITOR OK
2 EMIT L 3 EMIT
SCR # 101
( FORTH MNEMONIC ASSEMBLER )
( c 1982 J.W.Brown )
FORTH DEFINITIONS HEX FORGET TASK : TASK ;
: PUSH 4C C, 8240 , ; : PUT 4C C, 8242 , ; : NEXT 4C C, 8247 , ;
: SETUP 20 C, 8272 , ; : POPTWO 4C C, 835A , ;
: POP 4C C, 835C , ; : PUSH0A 4C C, 8545 , ;
VOCABULARY ASSEMBLER IMMEDIATE ASSEMBLER DEFINITIONS
: INS ( a new defining word for single byte instructions )
<BUILDS C, DOES> C@ C, ;
AA INS TAX 8A INS TXA A8 INS TAY 98 INS TYA BA INS TSX
88 INS DEY 68 INS PLA 4A INS LSRA 08 INS PHP 28 INS PLP
58 INS CLI 78 INS SEI 40 INS RTI 00 INS BRK 60 INS RTS
E8 INS INX C8 INS INY 2A INS ROLA 6A INS RORA CA INS DEX
48 INS PHA 0A INS ASLA 9A INS TXS 18 INS CLC 38 INS SEC
D8 INS CLD F8 INS SED B8 INS CLV EA INS NOP 20 INS JSR
4C INS JMP 6C INS JMI ( jump indirect ) -->


#( FORTH MNEMONIC ASSEMBLER )                                    0
SCR # 102
( FORTH ASSEMBLER CTD )
( Branch op-codes defined by same method as above )
90 INS BCC B0 INS BCS F0 INS BEQ 30 INS BMI
D0 INS BNE 10 INS BPL 50 INS BVC 70 INS BVS
: P ( artificial program counter ) HERE ;
: BTEST ( give msg if branch too large )
IF ." Branch" . ." too large" QUIT THEN ;
: >> ( make branch forward ) DUP 7F > BTEST 2 - C, ;
: << ( make branch back ) DUP 80 > BTEST 2+ 100 SWAP - C, ;
: BRTEST ( test instruction on stack, leave true if branch )
10 AND ;
: INSTR ( build instruction body for 9 values, on execution- )
( place the start address of this body on the stack )
<BUILDS 9 ALLOT DOES> ;
( an instruction body can now be defined  eg:- INSTR LDA etc )
( the relevant values installed in the instruction body ) -->
```

```
SCR # 103
( FORTH ASSEMBLER CTD )
INSTR LDA A1 B1 B9 00 BD B5 AD A5 A9 LDA 9 CFILL
INSTR STA 81 91 99 00 9D 95 8D 85 00 STA 9 CFILL
INSTR LDX 00 00 BE B6 00 00 AE A6 A2 LDX 9 CFILL
INSTR STX 00 00 00 96 00 00 8E 86 00 STX 9 CFILL
INSTR LDY 00 00 00 00 BC B4 AC A4 A0 LDY 9 CFILL
INSTR STY 00 00 00 00 00 94 8C 84 00 STY 9 CFILL
INSTR ADC 61 71 79 00 7D 75 6D 65 69 ADC 9 CFILL
INSTR AND 21 31 39 00 3D 35 2D 25 29 AND 9 CFILL
INSTR BIT 00 00 00 00 00 00 2C 24 00 BIT 9 CFILL
INSTR CMP C1 D1 D9 00 DD D5 CD C5 C9 CMP 9 CFILL
INSTR EOR 41 51 59 00 5D 55 4D 45 49 EOR 9 CFILL
INSTR ORA 01 11 19 00 1D 15 0D 05 09 ORA 9 CFILL
INSTR SBC E1 F1 F9 00 FD F5 ED E5 E9 SBC 9 CFILL
INSTR INC 00 00 00 00 FE F6 EE E6 00 INC 9 CFILL
INSTR DEC 00 00 00 00 DE D6 CE C6 00 DEC 9 CFILL ( ctd ) -->
```

```
SCR # 104
( FORTH ASSEMBLER CTD - cont of 9 value instruction bodies )
INSTR CPX 00 00 00 00 00 00 EC E4 E0 CPX 9 CFILL
INSTR CPY 00 00 00 00 00 00 CC C4 C0 CPY 9 CFILL
INSTR ROL 00 00 00 00 3E 36 2E 26 00 ROL 9 CFILL
INSTR ROR 00 00 00 00 7E 76 6E 66 00 ROR 9 CFILL
INSTR ASL 00 00 00 00 1E 16 0E 06 00 ASL 9 CFILL
INSTR LSR 00 00 00 00 5E 56 4E 46 00 LSR 9 CFILL
: BSIZE ( test for 1 or 2 bytes value on stack- leave true if 2 )
DUP FORTH FF00 AND IF 1 ELSE 0 THEN ASSEMBLER ;
: BERROR ( give error if operand too large ) CR IF
." Operand " U. ." too large" DROP QUIT THEN ;
: IERROR ( give error if illegal instr. )
CR ." Illegal instruction preceding " U. QUIT ;
: OPS! ( store operand in dictionary )
BSIZE IF , ELSE C, THEN ; ( next stores ins & operand in dict )
: INS! ROT + C@ -DUP IF C, OPS! ELSE IERROR THEN ; -->
```

SCR # 105

```
( FORTH ASSEMBLER CTD - COMPILER DIRECTIVES )
: # ( immediate mode ) BSIZE BERROR SWAP C@ -DUP IF C, C,
ELSE IERROR THEN ;
: , ( compile either abs or z.page address )
BSIZE 1+ INS! ;
: ,X ( compile ins & X indexed addr. ) BSIZE 3 + INS! ;
: ),Y ( compile ins & indirect post-indexed addr. )
BSIZE BERROR 7 INS! ;
: ,Y ( compile ins and Y indexed addr. ) BSIZE 5 + INS! ;
: ,X) ( compile ins and pre-indexed indirect addr. )
BSIZE BERROR 9 INS! ;
: F, FORTH , ASSEMBLER ; ( re-define FORTH "," as F, )
: CODE ( Mark a headerless primitive ) HERE 2+ F, ;
( This completes the first part of the Assembler )
( The next definitions are concerned with an active label )
-->
```

SCR # 106

```
( FORTH ASSEMBLER CTD - ACTIVE LABELS )
: PF2 DUP 2+ @ ; : BRBACK HERE 1 - SWAP @ - << ;
: BRFWD PF2 1+ OVER @ SWAP - SWAP 2+ @ C! ;
: JSTORE PF2 SWAP @ SWAP ! ; : JBACK @ F, ;
: GETIM HERE 1 - C@ ; : [X] GETIM BRTEST IF BRBACK ELSE JBACK
THEN ; : [Z] HERE SWAP 2+ ! GETIM BRTEST IF 1 ALLOT ELSE
2 ALLOT THEN ; : [XZ] DUP @ IF [X] ELSE [Z] THEN ;
: [Y] PF2 1 - C@ BRTEST IF BRFWD ELSE JSTORE THEN ;
: [RY] PF2 IF [Y] ELSE DROP THEN ;
: QLABEL SWAP IF DUP HERE SWAP ! [RY] ELSE [XZ] THEN ;
: LABEL <BUILDS 0 0 F, F, DOES> QLABEL ;
: LCLEAR -FIND IF DROP 2+ 0 SWAP 2DUP ! 2+ ! ELSE 0 MESSAGE
THEN ;
: ?LABEL -FIND IF DROP 2+ DUP @ U. 2+ @ U. ELSE 0
MESSAGE THEN ;
;S
```

```
SCR # 1
( BBC DEMO OF FORTH C 1982 J W Brown)
FORTH DEFINITIONS DECIMAL
: J RP@ 7 + @ ; : K RP@ 9 + @ ; ( define outer indices )
( FOURPOINT PATTERN - BASED ON BBC BASIC PROGRAM )
: DOPOINTS 500 0 DO I 500 - 0 MOVE 0 I DRAW 500 I - 0 DRAW
0 I -1 +- DRAW I 500 - 0 DRAW 15 +LOOP ;
: PAUSE ; : WAIT 32767 0 DO PAUSE PAUSE LOOP ;
: FOURPOINT 4 MODE 640 512 DGO DOPOINTS WAIT 8 0 DO 4 1 DO WAIT
3 I 0 DLC 0 J 0 DLC LOOP LOOP ;
-->
 OK
```

```
SCR # 2
( BBC DEMO OF FORTH C 1982 J W Brown )
FORTH DEFINITIONS DECIMAL ( CIRCLES ELLIPSES ETC )
: CIRCLES 5 MODE 5 EMIT 80 -80 D0 128 -128 DO 0 J J * I I *
+ 100 / 3 AND GCOL 69 J 80 + 8 * I 128 + 4 * PLOT
LOOP LOOP ;
: ELLIPSES 5 MODE 5 EMIT 80 -80 DO 128 -128 DO 0 J J * I I * +
J I * + 100 / 3 AND GCOL 69 J 80 + 8 * I 128 + 4 * PLOT
LOOP LOOP ;
: HYPERBOLAE 5 MODE 5 EMIT 80 -80 DO 128 -128 DO 0 J J * I I *
- 100 / 3 AND GCOL 69 J 80 + 8 * I 128 + 4 * PLOT
LOOP LOOP ;
: /HYPERBOLAE ( Bent Hyperbolae ) 5 MODE 5 EMIT
80 -80 DO 128 -128 DO 0 J I + J * I I * - 100 / 3 AND GCOL
69 J 80 + 8 * I 128 + 4 * PLOT LOOP LOOP ;
;S

OK
```

```
SCR # 3
( BBC DEMO OF FORTH C 1982 J W Brown )
FORTH DEFINITIONS DECIMAL ( Russian Roulette )
DEAD! 5 MODE 0 20 GCOL 100 600 MOVE 0 600 DRAW
0 0 DRAW 800 0 DRAW 800 150 DRAW 675 150 DRAW 675 0 DRAW
500 0 MOVE 500 275 DRAW 739 150 MOVE 739 155 DRAW ;

: INTRO ." You have a loaded revolver" CR
." pointed at your head !!" CR ." Do you want to pull the trigge
r ?" CR QUERY INTERPRET ;
: YES 5 ABSRND 4 = IF DEAD! ELSE CLT ." LUCKY YOU !" CR
INTRO THEN ;
: NO CLT ." CHICKEN !!!!" 10 0 DO 7 EMIT LOOP ;
;S



OK
```

```
SCR # 0
( BBC DEMO OF FORTH C 1982 J W Brown )
FORTH DEFINITIONS DECIMAL ( NIM )
: MATCHES 1+ 1 DO I 40 * 0 MOVE 0 2 GCOL I 40 * 300 DRAW
0 1 GCOL I 40 * 325 DRAW LOOP ;
: INTRO ." We take turns" CR ." to remove 1,2or3" CR
." matches-he who" CR ." takes the" CR ." last match-loses!" ;
: DMATCHES 12 ABSRND 21 + INTRO ;
: MCOUNT CR DUP . ." matches,your turn" ;
: TAKE? CR ." How many ?" QUERY INTERPRET ;
: TEST BEGIN TAKE? DUP 3 > >R DUP 1 < >R 2DUP < R> OR R> OR
WHILE DROP REPEAT ; : HTALLY DUP CR ." You took " . ;
: CTALLY CR ." I take " . ; : MCALC - DUP DUP 4 MOD DUP 1 =
IF 3 ABSRND + ELSE 3 + 4 MOD THEN - SWAP OVER - ;
: (NIM) 5 MODE DMATCHES BEGIN DUP MATCHES MCOUNT DUP 1 > WHILE
TEST CLT HTALLY MCALC REPEAT ; : NIM (NIM) DUP 0 < >R DUP
0= R> OR CR IF ." Oops I lose" ELSE ." I win!" THEN DROP ; -->
OK
```

ERROR HANDLING
(ABORT) and MESSAGE

In this implementation of FORTH, most detectable errors will result in an error message of the form:-

cccc ? MSG = n

cccc being the word where the system believes the error occured, and n a value representing the error type.
Most of these errors result in the clearing of both return and computation stacks, with the exception of error message 4, which is simply a warning.
When given, the error message number will be in the current numeric BASE, so the user should taken note of this whilst interpreting the error.
Below are given the error message numbers and their meaning.

DECIMAL	HEX	MESSAGE
0	0	Unrecognised word or symbol
1	1	Stack empty
2	2	Dictionary full
3	3	Incorrect address mode (for Assembler use)
4	4	Not unique (word already exists in dictionary)
5	5	Index or value outside valid range
6	6	Screen number outside valid range
7	7	Stack full
8 - 16 inclusive		for disc use and user defined
17	11	Compilation only
18	12	Execution only
19	13	Conditionals not paired
20	14	Definition unfinished
21	15	Protected dictionary
22	16	Loading use only
23	17	Off editing screen
24	18	Declare vocabulary

Major system failure is catered for by BRK vectors, which point to a warm start in FORTH.
The user may modify the system response to errors by substituting error handling procedures for those contained in (ABORT) and/or MESSAGE.
In the present system (ABORT) simply points to ABORT, and as (ABORT) is booted to location &492, the user can substitute the execution address of a new ABORT for the address stored in &49E.
Similary, an intersect vector for MESSAGE is provided at &502. In this case the error number is on the stack, and the users routine will be preceded by:-

? MSG # user-supplied-message

where the user routine can make use of the number on the stack. One such possibility is simply to print out the error messages in full, rather than simply the number, and this could be achieved by a positional-case definition, e.g.:-

PCASE ECASES   MSG0 MSG1 MSG2 .... etc ;

where MSG0 etc. refer to user supplied routines to print out the relevant message, and NULL does nothing (:NULL;)
The new MESSAGE pointed at from &502, could be as follows:-

: NEW-MESSAGE DUP (duplicate the error number)
(and print it)
CR ECASES ; (print out full message)

An example of this format in use for MSG 4 could be:-

cccc ? MSG # 4
Not unique

Of course, the above are suggestions only, and the user should take special care that any new system ABORT definition is correct, befpre re-vectoring the system.

APPENDIX B - Ascii codes

MSB

LSB	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	£	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL